Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Dissertação de Mestrado em Engenharia Informática

2º Semestre, 2008/2009

Paralelização de Algoritmos de Filtragem baseados em XPATH/XML com recurso a GPUs

João Manuel da Silva Araújo

Nº 26116

Orientador

Prof. Doutor Sérgio Marco Duarte

29 de Julho de 2009

Nº do aluno: 26116
Nome: João Manuel da Silva Araújo

Título da dissertação:

Paralelização de algoritmos de Filtragem baseados em XPATH/XML com recurso a GPUs.

Palavras-Chave:

- Paradigma editor/assinante
- algoritmos de filtragem
- paralelização
- gpu
- CUDA
- NVIDIA

Keywords:

- publish/subscribe paradigm
- matching algortithms
- paralelization
- gpu
- CUDA
- NVIDIA

Agradecimentos

Em primeiro lugar queria agradecer toda a dedicação e paciência que o meu orientador, Professor Sérgio Duarte, teve neste processo tão moroso e complicado mas muito importante da minha vida.

Queria também agradecer tudo o que a minha família fez por mim, principalmente à minha Mãe, Júlia Maria Ponce da Silva, que lutou e fez de tudo para que o realizar deste sonho fosse possível, Obrigado Mãe. Agradeço também ao meu irmão, Filipe Jorge da Silva Araújo por me ter ajudado nos momentos mais complicados e por me ter dado momentos de descontracção e de divertimento extraordinários, que se revelaram vitais ao longo da minha vida. Por último, ao meu Pai, Manuel Jorge Pereira de Araújo, que apesar de tudo, em algumas fases da minha vida teve um papel importante. Agradeço também ao meu avô e irmão, António Miranda da Silva e António Alexandre da Silva Santos, que também foram importantes para que eu atingisse esta fase da minha vida académica.

Não menos importante, agradeço todo o apoio que Janete de Freitas Martins me ofereceu, porque sem ela esta última fase da minha vida não seria realizada. Agradeço também todos os momentos especiais que me proporcionou porque foram dos mais importantes da minha vida.

Por último, uma palavra de agradecimento a todos os meus amigos e colegas que tive o privilégio de conhecer nesta fase, entre os quais destaco o Rui Miguel Moreno Rosendo, por tudo o que passámos. Obrigado também ao Rui Felizardo, Ricardo Salvador, Luísa Lourenço e Tiago Pessoa. Uma palavra também para todos os meus colegas do gabinete 254, do edifício 2 da FCT, que ajudaram-me a distrair em vários momentos difíceis.

A terminar, quero agradecer os 14 anos de companhia oferecidos por Garfi e Xina, que estiveram presentes nos momentos mais importantes, e porque foram os melhores gatos do mundo, descansem em paz.

Resumo

Esta dissertação envolve o estudo da viabilidade da utilização dos *GPUs* para o processamento paralelo aplicado aos algoritmos de filtragem de notificações num sistema editor/assinante. Este objectivo passou por realizar uma comparação de resultados experimentais entre a versão sequencial (nos CPUs) e a versão paralela de um algoritmo de filtragem escolhido como referência. Essa análise procurou dar elementos para aferir se eventuais ganhos da exploração dos *GPUs* serão suficientes para compensar a maior complexidade do processo.

Abstract

This thesis studies the feasibility of using *GPUs* for parallel processing algorithms applied to the filtering of events in a publish/subscribe system. This work involved a comparison between the experimental results of a sequential version (*CPU* version) and a parallel version of a referenced matching algorithm. This analysis has provided elements for arguing if any gains of using *GPUs* to perform fast matching of events are enough to offset the increased complexity of the process.

Índice

Lista de figuras

Lista de tabelas

1. Introdução

1.1. Introdução geral ou Motivação

A Internet é uma rede de comunicação que permite a troca constante de informação e dados entre vários intervenientes, localizados em qualquer parte do globo. A evolução da tecnologia tornou a Internet numa ferramenta importante para a obtenção de informação e partilha do conhecimento, hoje instanciada na *World Wide Web*. Os utilizadores acedem a páginas de conteúdos e retiram e/ou adicionam a informação do seu interesse. Para facilitar o acesso repetido a essa informação, normalmente, são utilizados *bookmarks* (favoritos), os quais ficam guardadas no *Web browser*, que não são mais que apontadores, denominados por *URL*, dos sítios que contêm a informação pretendida. Porém, na forma fundamental da *Web*, cabe ao próprio utilizador a iniciativa de monitorizar os seus sítios predilectos com vista a detectar potenciais actualizações. Com a expansão da Internet, a *Web* cresceu a um ritmo explosivo e mais e mais páginas e fontes de informação foram surgindo, aumentando grandemente o volume de informação existente e, por consequência, o número de *bookmarks* do seu interesse. Com esse aumento incessante, é já bastante difícil ao utilizador conseguir acompanhar e aceder a toda a informação potencialmente do seu interesse. Existe, assim, a necessidade de possibilitar ao utilizador estar a par de toda essa informação, de uma forma mais automática, e com tempos de busca mais reduzidos. Foi com este objectivo que surgiram os sistemas *RSS* [24].

RSS, abreviatura inglesa de *Really Simple Syndication*, é um mecanismo que tem, precisamente, como objectivo permitir ao utilizador estar mais facilmente sincronizado com as actualizações frequentes da informação Web, tais como *blogues*, jornais online, etc. Neste sentido, este mecanismo pode ser visto como uma extensão ao modelo original da *Web*.

Concretamente, os *RSS* foram criados com o intuito de ajudar os utilizadores a identificar e recolher, de uma forma mais fina, a informação que pretendem, através de uma codificação compacta em ficheiros *XML* [25], segundo um formato normalizado pela *W3C* [26]. Os ficheiros *RSS* que se encontram alojados nos mesmos servidores *Web* da informação a que dizem respeito correspondem, concretamente, a um resumo da informação recentemente disponibilizada ou actualizada, por exemplo, numa página ou *blogue*, oferecendo *hiperligações* directas para cada uma das actualizações, individualmente, ou seja, o *URL* correspondente. Para tal fim, existem várias possibilidades, entre as quais a utilização de aplicações com a finalidade de acederem aos servidores, e de filtrarem os ficheiros *RSS* alojados. Um exemplo prático pode ser um *add-on* do *Web Browser Mozilla* chamado de *Wizz*. A essas aplicações dá-se, normalmente, o nome de leitores de *RSS*, que podem ajudar o utilizador a focar-se em apenas um subconjunto da informação, podendo ainda também ajudar a filtrar os ficheiros consoante os

seus gostos e interesses. Esses leitores utilizam vários métodos para a eventual filtragem, como por exemplo, expressões regulares ou *XPATH* [27], que é uma linguagem de expressões também normalizada pela *W3C*. Devido à ideia de notificação inerente ao *RSS* e (eventualmente) à inclusão de filtragem de conteúdos, pode-se ver este mecanismo como sendo uma espécie de embrião de um paradigma editor/assinante [1] para a Web.

Em abstracto, o paradigma editor/assinante (*publish/subscribe*) é um sistema de intercomunicação envolvendo vários intervenientes, tipicamente num ambiente distribuído. Neste, existem editores (emissores) que publicam as mensagens, e que neste contexto frequentemente são chamadas de notificações ou eventos. Os assinantes (receptores) idealmente recebem apenas as mensagens do seu interesse, seleccionadas através dos temas escolhidos previamente, explicitados sob a forma de subscrições. No modelo editor/assinante não é necessário que os editores e os assinantes tenham conhecimento uns dos outros. O paradigma desacopla, assim, o emissor (editor) do receptor (assinante), podendo tornar o sistema mais flexível, permitindo interacções separadas não só no espaço como no tempo. O sistema também é, assim, compatível com um modelo de comunicação assíncrona, onde não há o bloqueio quer no envio de uma mensagem por parte do emissor quer na recepção do receptor.

Existem, essencialmente, duas classes distintas deste tipo de sistemas: os sistemas editor/assinante *baseados em tópicos* e os *baseados no conteúdo*. Nos que são *baseados em tópicos*, as mensagens podem estar agrupadas em grupos, tópicos ou canais. Os seus editores publicam as mensagens num canal e são responsáveis por definir os tópicos existentes. Cada assinante pode subscrever um desses tópicos através de subscrições, recebendo apenas as mensagens desse tópico. Para maior eficiência e conveniência surgiu a necessidade de ir mais longe e permitir uma filtragem semântica dos eventos. Os sistemas que permitem tal funcionalidade são designados sistemas *baseados no conteúdo*. Os sistemas editor/assinante *baseados no conteúdo* enquadram-se numa problemática mais ampla denominada *encaminhamento baseado no conteúdo* [2, 3].

Em muitos sistemas editor/assinante é introduzido um intermediário, entre os editores e os assinantes, tipicamente designado *broker*. O *broker* tem como principal objectivo ser o ponto de encontro entre editor/assinante e receber e encaminhar as mensagens introduzidas no sistema. Assim, é delegada a tarefa de gerir as mensagens do sistema para um interveniente independente dos restantes, havendo o desacoplamento dos editores dos assinantes, não sendo necessária a sua presença simultânea ou conhecimento mútuo. O *broker* também pode ter a responsabilidade de filtrar as mensagens dos editores, encaminhando apenas os eventos do interesse dos assinantes, no caso de o encaminhamento ser baseado no conteúdo.

A filtragem torna-se assim um ponto fulcral nos sistemas editor/assinante baseados no conteúdo, em especial nos sistemas baseados em *brokers* onde o número de assinantes é usualmente muito significativo. Como o número de subscrições é proporcional ao número de

assinantes, e como a taxa de entrada de novos eventos nestes sistemas é geralmente muito elevada, para os *brokers* torna-se importante a eficiência e rapidez de obtenção do resultado da filtragem. Essa rapidez poderá ser obtida através de um algoritmo de filtragem particularmente eficiente ou, então, através da paralelização de um algoritmo mais genérico. A grande importância dos algoritmos de filtragem advém do débito do sistema depender directamente da capacidade de realizar a filtragem em tempo útil. Um algoritmo de filtragem eficaz e eficiente pode ainda trazer benefícios a nível de poupança nas necessidades de largura de banda, ao evitar o envio de eventos não desejados já que esses seriam filtrados de uma forma mais fina, evitando os falsos positivos no sistema e por consequência haverá menos ocupação da largura de banda, contribuindo assim para uma maior escalabilidade do sistema.

O objectivo desta dissertação centra-se na optimização do processo de filtragem, ou seja, em estudar vários algoritmos de filtragem e procurar o seu melhoramento através da paralelização de um algoritmo adequado. Especificamente, procura-se estudar essa possibilidade com recurso a *GPUs* [28] das placas gráficas.

O interesse dessa escolha deve-se à natureza altamente paralela dos *GPUs* correntes. A evolução recente, tanto a nível do aumento do poder de processamento como a nível do maior paralelismo oferecido, tornou as placas gráficas numa solução atractiva a ser explorada para processamento genérico. Esta evolução veio acompanhada por um novo paradigma de programação genérica, de nome *General-Purpose Computation on Graphics Processing Units*, ou *GPGPU*. Este paradigma foi proposto e aplicado em várias áreas de investigação, como por exemplo o *Folding@Home* [36]. Nesse sentido, foram apresentadas, pelos vários fabricantes de placas gráficas, várias *APIs*, de entre as quais se destacam o *CUDA* [31], *API* proprietária da *NVIDIA*, e o *OpenCL* [32, 33], uma *API* que oferece a possibilidade de programação nos *GPUs* independentemente do proprietário do *hardware*, sendo proposto pela *Apple/Khronos Group*.

Além do elevado paralelismo, estes *GPUs* caracterizam-se por uma performance elevada nas operações de vírgula flutuante. Nesta dissertação essa faceta não é explorada, porém há ainda a maior largura de banda disponibilizada pelas memórias dos *GPUs*, que em alguns casos chega a ser dez vezes maior, comparativamente com *CPUs* convencionais. [4]. À luz dessas características, juntamente com a redução do seu custo, os *GPUs* aparentam ser uma escolha atractiva para resolver o problema da filtragem, transformando-se numa possível solução mais barata do que os processadores convencionais, na relação a desempenho/preço.

Em suma, esta dissertação centra-se num estudo da viabilidade da utilização dos *GPUs* para o processamento paralelo aplicado aos algoritmos de filtragem de eventos num sistema editor/assinante. Este objectivo passa por realizar uma comparação de resultados experimentais por entre várias versões paralelas de um algoritmo de filtragem escolhido entre vários. Essa análise procura dar elementos para uma discussão da real possibilidade da utilização dos *GPUs* e se compensará a sua utilização neste âmbito face à complexidade do processo. Num âmbito

mais secundário, os eventos de referência são ficheiros *RSS* e a sua filtragem é feita com recurso a expressões *XPATH*.

1.2. Principais Contribuições

As principais contribuições são as seguintes:

- Produzir um estudo e uma análise da viabilidade dos *GPUs* para o processamento de dados em sistemas editor/assinante *baseados no conteúdo*.
- Estudo das adaptações efectuadas no algoritmo escolhido como referência.
- Estudo experimental e crítico da implementação efectuada para provar se existirão ganhos ou perdas na utilização dos *GPUs* no processamento das notificações e subscrições do sistema.

1.3. Organização

Após este capítulo, onde se fez a apresentação do contexto deste trabalho, o documento prossegue da seguinte forma: no capítulo 2 apresenta-se um resumo do estado da arte, organizado em três secções que descrevem os ficheiros *RSS*, os *GPUs* e por fim os principais algoritmos de filtragem estudados. No capítulo 3 apresenta-se toda a implementação efectuada, tanto do lado do *CPU* como do *GPU*. No capítulo 4 serão discutidos os resultados obtidos em todas as implementações efectuadas. Finalmente, no capítulo 5 apresentam-se as conclusões da dissertação.

2. Trabalho relacionado

Neste capítulo é feito um apanhado do estado actual dos principais pontos de foco da dissertação. Vão ser brevemente abordados os aspectos relacionados com o formato dos ficheiros *RSS*, que na dissertação correspondem aos eventos de um sistema editor/assinante, e serão analisados vários conjuntos de possíveis expressões *XPATH* para sua filtragem, que são consideradas as subscrições.

Será feito um resumo do estado actual dos *GPUs*, como a sua arquitectura e estrutura. Possíveis *APIs* para serem aplicadas na sua programação também serão apresentadas, sendo feita uma apreciação crítica destas e de qual foi a escolhida para ser aplicada.

Por último, são apresentados alguns dos algoritmos possíveis de serem aplicados no paradigma apresentado. A sua possível viabilidade em programar em paralelo também será alvo de estudo, sendo que também é feita uma análise da possibilidade de serem programados nos *GPUs*. O algoritmo escolhido como o melhor para ser paralelizado será alvo de um estudo mais aprofundado.

2.1. *RSS*

Really Simple Syndication (*RSS)* está em grande desenvolvimento [1], levando ao surgimento de duas versões do formato. A última versão é a 2.0 *RSS*, que consiste num ficheiro *XML*, normalizado segundo a especificação da versão 1.0 publicada no website de *World Wide Web Consortium (W3C)*. Um ficheiro *RSS*, também denominado como *feed*, num sistema editor/assinante é um conjunto de eventos ou notificações introduzidas e disponibilizadas aos assinantes. Oferecem a possibilidade de escolha e obtenção dos temas de maior interesse, através de subscrições. Por exemplo, o sistema Cobra [5] demonstra quais os possíveis ganhos obtidos na utilização desta ferramenta.

Na dissertação, as subscrições poderiam ser representadas segundo uma linguagem normalizada, a *XPATH*. Essa, como *XQUERY* [29], são linguagens de expressões desenvolvidas para a filtragem de ficheiros *XML*, como tal, poderiam ser a melhor escolha para esse efeito. Os resultados obtidos pelas expressões *XPATH*, que poderão ser consideradas como filtros, indicarão quais os subscritores interessados nos novos eventos, à medida que estes chegam. Como o algoritmo foi desenvolvido de modo a ser genérico, uma conversão das expressões facilita a obtenção da filtragem.

Nas duas secções seguintes serão introduzidos dois pontos fundamentais: a estrutura dos ficheiros *RSS* e os possíveis filtros. Na primeira secção, são apresentados os principais elementos a serem utilizados para filtragem e de seguida são introduzidas as possíveis expressões *XPATH* a serem aplicadas.

2.2. *XML*

O *XML* que define um *RSS* é relativamente estático e rígido na sua estrutura, tendo sempre como elemento raiz <*rss*>. Esse terá um atributo indicativo da versão *RSS* utilizada (denominado *version*). O único elemento filho da raiz é <*channel*>, que contém filhos <*item*> *(*são estes que contêm nos seus filhos a informação pertinente a cada evento quando considerado individualmente) e outros elementos que caracterizam a *feed*, como a data de submissão, autor, título, *hiperligação*, etc…. Os filtros *XPATH* serão aplicados na informação dos eventos, porque é aí que reside toda a informação útil, como a data de submissão, autor, url, descrição, título, etc…, para a obtenção dos eventos de interesse do assinante. É ainda de salientar que os documentos *RSS* são tipicamente de pequena dimensão e, essencialmente, correspondem a uma lista de itens.

Na dissertação, um evento é considerado como uma lista de atributos da forma (elemento, valor). Fazendo a analogia com a representação apresentada dos eventos (conjunto de elementos filhos de <item>), os atributos teriam a seguinte forma: (title, "título da notícia"), (description,"descrição da notícia"), etc. Assim, para simplificação do tratamento, poderá facilitar desdobrar um documento *RSS* em vários eventos independentes.

Para a sua utilização como *input* dos algoritmos de filtragem nos *GPUs*, surge a necessidade da tradução destas em *strings*. O processo para essa tradução é discutido na secção *implementação*.

2.3. *XPATH* (Filtros)

Como indicado anteriormente, os filtros focam-se tipicamente nos elementos filhos de <*item*>. Como a filtragem é realizada para a obtenção de todos os eventos do interesse dos assinantes, as expressões serão fundamentalmente iniciadas com "//item". Esta expressão devolve todos os eventos, mas para uma filtragem mais fina, existem os operadores lógicos oferecidos pelo *XPATH*, como o *and*, *or* ou os operadores relacionais (=, >, <, …). Estas expressões representam as subscrições, que podem ser consideradas como um conjunto de predicados da forma (elemento, operador de comparação, valor). Por exemplo a expressão //item[title = "título"] pode ser interpretada como equivalente a um predicado da forma (title, =, título). Mais uma vez emerge o problema subjacente da utilização das *strings* nos *GPUs* que mais uma vez é indicado na secção *implementação*. De realçar que no trabalho da dissertação não foram utilizados filtros (ou subscrições) com o formato *XPATH* mas como o algoritmo é genérico facilmente através de uma tradução das expressões se obtém o formato correcto.

2.4. *GPUs*

Acompanhando a evolução da indústria dos vídeo jogos, os *GPUs* tiveram um grande desenvolvimento no seu poder de processamento [6]. Inicialmente, os *GPUs* eram limitados pela sua própria arquitectura e eram apenas direccionados para o processamento gráfico. Com o aumento da exigência do nível de detalhe gráfico e da variedade de efeitos a suportar, o processamento em paralelo tornou-se a melhor solução para um aumento da oferta de poder de processamento. Além da arquitectura dos *GPUs* se ter transformado numa natureza mais paralela, houve uma evolução importante no modo como os efeitos gráficos passaram a ser obtidos.

Para a possibilidade de uma maior flexibilidade na complexidade dos gráficos, foram acrescentadas novas funcionalidades a nível da programação, sob a forma de *shaders* [30]. Os *shaders* que consistem em pequenos programas especializados e executados dentro do processador gráfico, revolucionaram a indústria dos *GPUs*. Com essa possibilidade de processamento mais genérico, os *GPUs* passaram a poder ser explorados noutros domínios da computação intensiva, não necessariamente relacionada com processamento gráfico, tal como a simulação de fenómenos físicos. Este novo paradigma de programação é denominado de *GPGPU* [7]. *GPGPU*, que significa *General Purpose Graphic Processing Unit*, consiste na utilização dos *GPUs* para computação de aplicações de índole geral. Esta evolução foi acompanhada pela introdução das primeiras *APIs* específicas para explorar este novo modelo de programação. Um facto menos positivo é que, até à data, essas *APIs* são proprietárias e incompatíveis entre si, e apenas funcionam na mesma linha de dispositivos de cada fabricante, ex., *NVIDIA* ou *ATi*. Recentemente, surgiu um esforço para introduzir uma solução independente da plataforma, tal como se explicará mais à frente.

Nas próximas secções serão tratados os pontos fulcrais dos *GPUs* recentes, como a sua arquitectura, e será feito um levantamento das principais *APIs* e como elas se podem conjugar com o problema apresentado, ou seja, a possibilidade de paralelização dos algoritmos.

2.4.1. Arquitectura

Um *GPU* pode ser visto como um dispositivo de computação autónomo, ou então como um co-processador para o *CPU* (*host*) [8]. Para uma melhor computação paralela e distribuída, cada *GPU* contém a sua memória local que se caracteriza por ter uma elevada largura de banda (comparativamente ao *CPU*) [4]. Internamente, cada *GPU* é composto por vários processadores, cada um deles pode correr vários *threads* em paralelo, com a característica desses conseguirem cooperar e comunicar entre si com baixo *overhead*, partilhando um espaço de memória eficientemente e com baixa latência. Apesar de nos *CPUs* também haver possibilidade de

utilização de *threads*, as principais diferenças residem no facto de a criação desses terem um *overhead* consideravelmente menor do que o acontece nos *CPUs* [10].

Com essas características, o poder que se pode retirar destes dispositivos é imenso, isto porque os *GPUs*, ao contrário dos *CPUs* convencionais, têm uma arquitectura nativa paralelizada, ou seja, cada *Core* de um *GPU* pode correr centenas de *threads*, e com o conjunto dos vários *Cores* existentes o número de *threads* a executarem em simultâneo pode atingir os milhares. Por exemplo, o poder de processamento de *GPUs* das placas gráficas da *NVIDIA* está na ordem de 50 a 200 *GFLOPS* (nas séries 8) [4, 8].

Tipicamente, cada *GPU* contém um conjunto de *Streaming Multiprocessors (SM) multithreaded*. Um *SM* consiste num conjunto de oito *cores Scalar Processors (SP)*. Cada *SM* terá um conjunto de *threads* associados que serão distribuídos pelos vários *SP* existentes de uma forma eficiente, explicada na secção seguinte.

A nível de programação no *GPU*, cada porção paralela de dados de uma aplicação é processada sob a forma de um *kernel*. Para o dispositivo, essa porção pode ser vista como um processo, onde poderão correr em paralelo vários *threads*. A seguir serão apresentadas as características principais dos *GPU NVIDIA*, de as quais se destacam: a organização dos *threads* e os vários tipos de memórias existentes.

2.4.2. Organização dos *threads*

O *SM* mapeia os *threads* para os *SP* existentes, sendo que cada *thread* têm uma execução independentemente das outras, tendo os seus próprios registos e endereços de instruções. Cada *SM* gere o conjunto de *threads* através de uma nova arquitectura introduzida neste novo tipo de *hardware*, *a Single Instruction, Multiple Thread (SIMT)*. *SIMT* é uma unidade que cria, organiza, escalona e executa os *threads* em *warps*. Os *warps* são grupos de 32 *threads* criados a nível de hardware, sendo lançados para serem executados um de cada vez em cada *SM* existente. O escalonamento dos *warps* é feito ao nível do *hardware* com *overhead* nulo, sendo essa uma das características que torna o lançamento de milhares de *threads* uma solução a explorar. Isto, porque a latência de acessos à *memória global* (que vai ser mais à frente indicado) de um *warp* proporciona a possibilidade de execução de outros *warps* enquanto esse executa a leitura, sendo a troca de contexto feita, como indicado, com *overhead* zero, tornando a execução bastante rápida.

A informação dos *threads*, que estão a executar, é mantida a nível de *hardware*, sendo necessários recursos desse. Tal facto impossibilita que o número total de *threads activos* no *SM* seja maior do que um determinado número, discutido no capítulo 4.

Sempre que é executado um *kernel* é necessário definir qual o tamanho da execução. Os *threads* estão organizados logicamente em *blocos*. Os *blocos* estão organizados numa *grelha*.

Uma *grelha* pode ir até três dimensões, perfazendo assim N*K*L *blocos* de *threads*. O *tamanho da execução* consiste, portanto, no tamanho da *grelha*. Esse *tamanho* é caracterizado pelo número de *blocos* e pelo número de *threads*, sendo o *tamanho da execução* definido por:

*Tamanho dos blocos * Número de threads*

Na figura seguinte está exemplificado como podem estar organizados os *threads* na execução de um kernel:

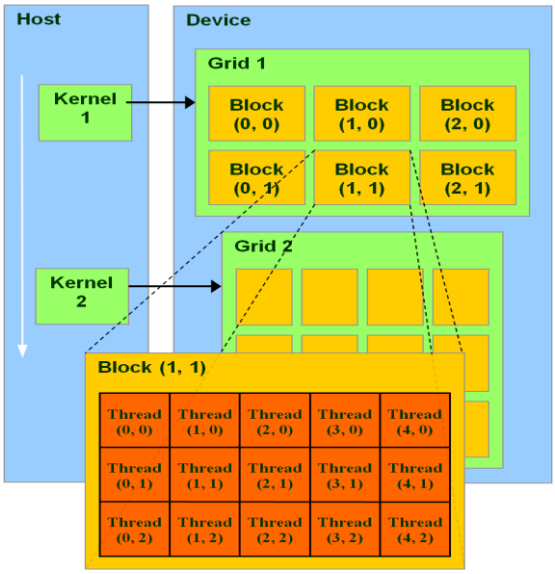


Figura 2.1 Organização do tamanho de execução de um *kernel*
(Figura retirada de [8])

Como se pode verificar pela figura 2.1, o *host* (*CPU*) cria um *kernel* para cada fluxo de execução, *kernel* esse que não é mais que uma *grelha* de *blocos* de *threads* com código associado. Cada *bloco* tem um identificador implícito, tal como os *threads*, e é através desses identificadores que se pode, no próprio *kernel*, definir que parte dos dados cada uma deverá processar, relacionando os identificadores com o método de indexação das estruturas de dados. Esses identificadores dos *blocos* são globais e únicos entre todos os lançados. No caso dos identificadores dos *threads*, são únicos localmente a cada *bloco*. Os identificadores de cada *bloco* e *thread* podem ser até um máximo de três dimensões, por exemplo (1) ou (1,2) ou até mesmo (1,2,3), isto apenas no caso dos *threads* devido às limitações impostas pelo *hardware*. Essas limitações serão apresentadas mais à frente no capítulo 4.

Em relação à organização em *warps*, cada *bloco* tem um conjunto de *warp* associados, sendo formado mais do que um se o número de *threads* pedidos o permitir. Ou seja, se por exemplo o tamanho do *bloco* for 96 *threads*, são formados três *warps* distintos. Os identificadores dos *threads* são organizados consecutivamente, ou seja, por exemplo na configuração 3 *blocos* e 96 *threads*, os *threads* são organizadas em três *warps*, onde o primeiro

terá os *threads* com os identificadores de 0 até 31, os 32 até 63 no segundo *warp*, e as restantes são incluídas no último *warp*. Na figura seguinte está exemplificada uma possível organização dos *threads*:

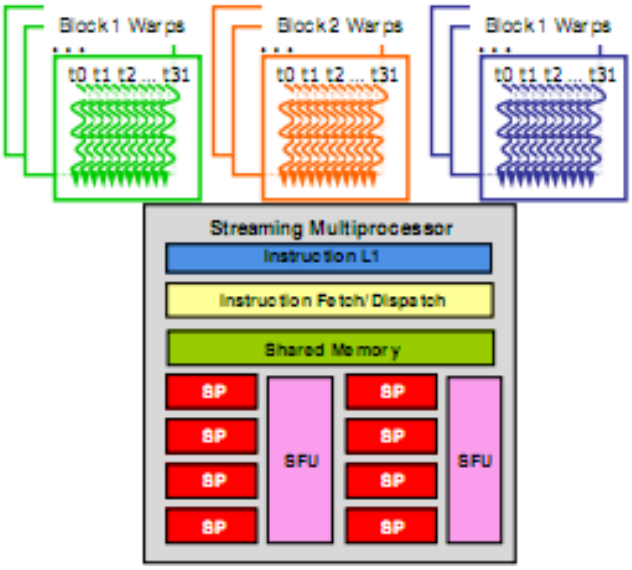


Figura 2.2 Organização dos *warps*
(Figura retirada de [8])

Na figura 2.2 estão representados três *blocos*, divididos em *warps* e distribuídos pelo *SM*. Como indicado anteriormente, o *SM* contém toda a informação dos *threads* que correm nele.

A seguir será feita uma breve apresentação das memórias disponíveis e alguns dos prós e contras do seu uso.

2.4.3. Memórias

O *GPU* tem basicamente cinco memórias distintas que podem ser acedidas pelos *threads* durante a sua execução. No contexto da dissertação apenas quatro são importantes, a *memória global*, a *memória partilhada* por cada bloco, a *memória local* a cada *thread* e a *memória constante*, sendo a restante, a *memória de texturas*, mais direccionada para processamento gráfico. A principal, ou pelo menos a que contém mais espaço, é a *memória global*. Principal porque é o principal espaço de comunicação e de troca de dados entre o *host* e os *threads*, podendo ser considerada a memória partilhada por entre todos os *blocos* de uma *grelha* porque é visível a todos os *threads* invocados. Esta memória tem o custo de os acessos serem realizados com a maior latência, em relação às outras, por isso a sua utilização deve ser o mais possível evitada. Tal acontece porque a memória não é *cached*. Este tipo de memória é persistente, ou seja, mantém-se através da chamada dos vários *kernel* lançados na aplicação, podendo assim ser possível utilizar várias possibilidades entre *kernel*, como por exemplo, existir

um que escreve um resultado na memória e outro que de seguida utiliza o resultado obtido para executar algo completamente diferente.

A *memória partilhada* é o espaço partilhado por todas os *threads* existentes num *bloco*. Aqui apenas os *threads* de cada *bloco* individual têm acesso aos dados localizados nele. Pode-se considerar a memória *individual* de cada *bloco*. Como a *memória partilhada* é "*on-chip*", com um comportamento análogo à cache *L1* dos processadores convencionais, o seu acesso, tanto para escrita como para leitura, é bem mais rápido do que acontece com a *memória global*. A diferença pode atingir 1/100 ciclos de relógio [4], tornando-se portanto um bom recurso a explorar, mas com a limitação de ser uma memória não persistente e por ser apenas partilhada entre cada *bloco* individualmente, não permitindo a partilha de resultados entre outros *blocos* de diferentes execuções. O seu tamanho é sempre de 16 *Kbytes* por *SM*.

A *memória local* é o espaço associado a cada *thread*. Nessa memória estarão todas as grandes estruturas que podem consumir um grande número de registos e que não são possíveis de serem colocadas nos registos individuais de cada *thread*.

Por último, a *memória constante* tem um funcionamento diferente das demais. Esta memória é *cached*, ou seja, um acesso para leitura à memória tem o mesmo custo que uma leitura na *memória global*, mas, se os *threads* seguintes acederem à mesma posição o acesso pode ser tão rápido como uma leitura dos seus próprios registos. Porque, como foi dito, o valor já se encontra *cached*. A *memória constante* tem disponível 64 *Kbytes* de espaço livre, mas, para o espaço de dados *cached* contém 8 *KBytes*.

Em relação às permissões permitidas a cada *thread*, existem permissões de escrita (W) e leitura (R) nas várias memórias do *GPU*. A seguir são indicadas essas permissões, descriminadas por cada memória:

- R/W nos seus próprios registos
- R/W na sua *memória local*
- R/W na *memória partilhada* do seu *bloco*
- R/W na *memória global*
- R na *memória constante*
- R na *memória de texturas*

O *host* (*CPU*) pode escrever e ler da *memória global*, *constante* e de *texturas*. Os registos também têm uma importância fundamental na execução de aplicações no *GPU*. Essa importância é mencionada e analisada no capítulo *validação experimental*.

Na figura seguinte está exemplificada a arquitectura existente:

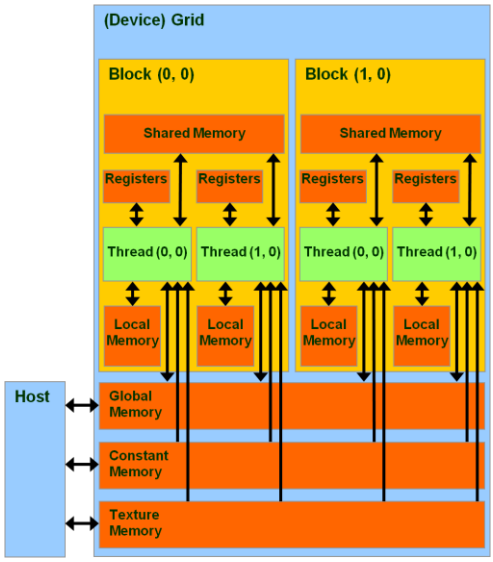


Figura 2.3 – Organização interna de uma *grelha*
(Figura retirada de [8])

De uma forma resumida, a arquitectura dos *GPUs* está altamente paralelizada, tornando possível o seu uso para o processamento de grandes volumes de dados. A utilização de *blocos* de *threads* oferece uma transparência da organização dos *threads* ao programador, tornando assim mais fácil a programação desses. A única limitação apresentada consiste na pouca *escalabilidade* dos *blocos* de *threads*, ou seja, apenas é possível lançar um determinado número fixo de *blocos* por ***cada*** execução (*kernel)* no *GPU*, estando esses limitados às características de cada placa gráfica. Na secção 2.4.4.6 estão descritas todas as especificações de computação oferecidas.

2.4.4. *APIs (CUDA)*

Compute Unified Device Architecture [28], *ou CUDA*, é a *API* proprietária da *NVIDIA* para uso exclusivo nos seus *GPUs*. Trata-se de uma extensão ao *ANSI-C* com suporte para novos tipos de dados, com primitivas de baixo nível para sincronização e acesso e gestão da memória dos *GPUs*. Existe uma *API* semelhante, para dispositivos *ATi*, mas incompatível com esta. Por iniciativa da *Apple*, está em desenvolvimento uma plataforma, denominada *OpenCL* (*Open Computing Language)* [32, 33], que visa permitir a exploração dos *GPUs* de uma forma mais genérica e com suporte para *hardware* heterogéneo. Um esforço semelhante existe no *DirectX11* [34], da *Microsoft*, que oferecerá também a possibilidade de execução de programas independentemente da plataforma de *hardware*.

Nesta dissertação, a *API* utilizada para implementação foi o *CUDA* porque é, à data, a melhor documentada.

O *CUDA* oferece um modelo de programação genérico ao utilizador, onde se pode tirar o melhor partido das capacidades dos *GPUs* para cálculos, tanto gráficos como para outras aplicações, oferecendo ferramentas de desenvolvimento que permitem aos utilizadores usarem uma variante de *ANSI-C*. O contexto inicial de utilização dos *GPUs* centrava-se nos gráficos/jogos, portanto o *CUDA* veio quebrar essa especificidade permitindo alargar os domínios de aplicação através desta *API*. Seguidamente, será feita uma apresentação sistemática das principais características do *CUDA*, das quais se destacam*:* Tipos de Dados, Declaração de Métodos, *Kernels*, Gestão de Memória e Comunicação *Host/Device*.

2.4.4.1. Tipos de Dados

Uma das extensões ao *ANSI-C que o CUDA* oferece consiste num novo conjunto de novos tipos nativos, como *int2, int3, int4, double2, double3, double4*, etc…que possibilitam, em apenas uma variável, ter mais do que um *inteiro, double*, etc... Foram implementados estes novos tipos para a facilidade de utilização novas estruturas que representassem pontos num espaço cartesiano, facilitando assim a sua utilização para cálculos matemáticos de processamento gráfico.

A título de exemplo, e no contexto da dissertação, um predicado com identificador geral no sistema igual a 3, operador de comparação "=" que no sistema pode ser representado por 0, e com valor 10, poderia ser representado num *int3* da forma:

Inteiro.x = 3
Inteiro.y = 0
Inteiro.z = 10

Provando, assim, ser um tipo nativo de grande utilidade para a implementação do algoritmo proposto na dissertação. Também existe a possibilidade de definição deste tipo através da utilização de um tipo *dim3*, que tem grande utilidade para a definição do tamanho da *grelha* e do número de *threads* pelos *blocos.*

Na implementação de um *kernel*, é necessário saber qual a posição da estrutura a que cada *thread* acede. Tal é possível através das variáveis *blockIdx* e *threadIdx*, que são do tipo *uint3*, e são variáveis *built-in* do *CUDA*. A dimensão de cada *bloco* também é acessível através da variável do tipo *dim3*, *blockDim.* Todas estas variáveis encontram-se na *memória partilhada* de cada *bloco*.Na dissertação, a utilização destas variáveis foi crucial, porque a indexação foi realizada através da sua utilização, sendo que através destas se obtém o identificador global, na execução, de cada *thread*, através da fórmula:

*blockIdx.x*blockDim.x + threadIdx.x*

Aqui, supõe-se que os recursos pedidos (número de *threads* e de *blocos*) para serem executados foram todos de apenas uma dimensão.

Para a declaração de variáveis existem vários qualificadores. Os qualificadores indicam em que memória residirá a variável declarada. O qualificador *__device__* indica que reside na *memória global* sendo o seu tempo de vida igual ao da aplicação, ou seja, é uma memória persistente. O qualificador *__constant__*, como o nome indica, refere-se a variáveis que residem em *memória constante*, tendo as mesmas características que as variáveis *__device__*. Por último, o qualificador *__shared__* é um qualificador que apenas é utilizado no código dos *kernel* porque define uma variável em *memória partilhada*, memória essa que apenas é acedida por *threads* do mesmo *bloco*.

2.4.4.2. Declaração de Métodos

Para a declaração dos métodos criados pelo programador existem regras no *CUDA*. Para métodos que podem ser utilizados apenas no dispositivo existe o qualificador *__device__*. Para métodos no *host* temos *__host__* que podem apenas ser executados e chamados no *host*, sendo que a omissão deste qualificador tem o mesmo significado. O qualificador *__global__* define um *kernel*, que como foi mencionado antes, consiste no código a ser executado pelos *threads*. Este método tem de devolver, *obrigatoriamente*, *void*. Como a *memória global* é persistente, os dados que são transferidos mantêm-se até ao fim da execução da aplicação desenvolvida.

De seguida é apresentada uma tabela com todas essas regras:

Declaração	Executado no	Chamada por	Tempo de vida
__device__ float DeviceFunc()	dispositivo	dispositivo	Aplicação
__global__ void KernelFunc()	dispositivo	host	Aplicação
__host__ float HostFunc()	host	host	Aplicação

Tabela 2.1 – Tabela de regras para métodos

2.4.4.3. *Kernels*

Um *kernel* tem uma sintaxe pré-definida para a sua execução. É necessária a indicação do número de *blocos* e *threads* para cada execução, o chamado *tamanho de execução*. A sintaxe será:

kernelName<<<NumberBlocks, NumberThreadsPerBlock>>(argumentos);

Pode também haver um terceiro argumento na configuração do *kernel*. Esse argumento tem como objectivo a indicação de qual será o tamanho da *memória partilhada* utilizada no *kernel* por cada *bloco*.

Para a utilização de variáveis na *memória partilhada* dos *blocos* é necessária a utilização de um qualificador denominado *__shared__* como indicado anteriormente. Estas variáveis são voláteis, significando que quando termina a execução do *kernel* os dados perdem-se. Mas, com a introdução do terceiro argumento, um novo indicador é necessário, o *extern*, que tem como objectivo indicar que o tamanho da variável utilizada, na *memória partilhada*, é indicado na configuração de lançamento do *kernel*.

Um dos métodos mais importantes oferecidos pelo *CUDA* é o ***__syncthreads().*** Tem como objectivo ser uma barreira de sincronização de *threads* do mesmo *bloco* num dado ponto no código, ou seja, garante a sua coordenação. Devido ao facto de terem de ser do mesmo *bloco*, não é possível sincronizar *threads* de *blocos* diferentes, sendo essa uma grande limitação.

2.4.4.4. Gestão de Memória

Com a necessidade de alocação e libertação de memória, a *API* necessariamente oferece vários métodos para tal fim. Como no *ANSI-C*, o programador tem que definir o que pôr, onde pôr, e que tamanho terá os dados que pretende utilizar.

Seguidamente, são apresentados os principais métodos para a gestão de memória, apenas utilizados no lado do *host*.

Método	Objectivo
cudaMalloc(Estrutura, Tamanho)	Tem como objectivo alocar espaço linear (contíguo) na *memória global*. Na secção de memórias é indicado quem o pode aceder.
cudaFree(Estrutura)	Serve para libertar o espaço alocado para a estrutura indicada no argumento. Função que liberta espaço na *memória global*.

Tabela 2.2 – Métodos fundamentais para a gestão de memória

O *CUDA* oferece outros métodos, como o *cudaMallocPitch* ou *cudaMalloc3D*, mas não foram utilizados neste trabalho. Para a *memória constante* não existe a necessidade de alocação de espaço.

2.4.4.5. Comunicação *host/device*

O único meio de comunicação entre o *host* e o *device* é a *memória global*, funcionando como uma memória partilhada entre eles. Nela, o *host* introduz todo o trabalho que o *device* tem de processar. Os resultados do processamento também são escritos nela. Para a transferência de dados entre o *host* e a *memória global*, ou vice-versa, é utilizado o método **cudaMemcpy().** Este método tem como objectivo a transferência para qualquer espaço de memória alocado. O método necessita de quatro parâmetros: o que se pretende transferir e para onde se quer transferir, o número de *bytes* copiados e o sentido de transferência de dados.
Existem quatro sentidos:

host - host

host - dispositivo

dispositivo - *host*

dispositivo – dispositivo

A transferência pode ser assíncrona, ou seja, a fonte não necessita de uma resposta do destinatário.

A figura seguinte é representativa da zona onde a função **cudaMemcpy()** intervém, ou seja, em quais das memórias, já previamente indicadas, este método pode realizar as transferências. Pode ser observado dentro da área assinalada pela seta apresentada de seguida:

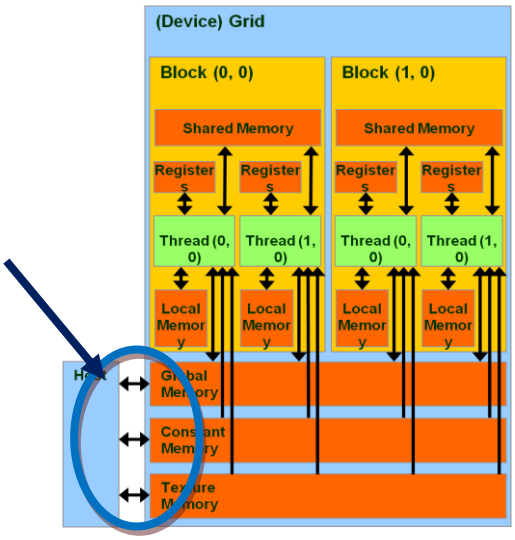


Figura 2.4 - Zonas de transferência de cudaMemcpy
(Figura retirada de [8])

No exemplo da transferência de uma matriz, existente no *host*, a linha de código seria:

cudaMemcpy(Dispositivo.Matriz, Host.Matriz, size, cudaMemcpyHostToDevice)

Na linha anteriormente apresentada a transferência está a ser realizada no sentido *host – dispositivo*. Pode-se retirar tal ilação através do último argumento, que indica o sentido de transferência.

Para a transferência de dados do *host* para a *memória constante* é utilizado outro método. Esse consiste no ***cudaMemcpyToSymbol().*** Os parâmetros são os mesmos que os do anterior método, mas existe uma diferença crucial, a estrutura para onde se copia os dados do *host* não tem que ser previamente alocada, sendo apenas inicializada com o qualificador *__constant__*. Um exemplo prático pode ser o apresentado no manual de programação do *CUDA* [4]:

```
__constant__ float constData[256];

float data[256];

cudaMemcpyToSymbol(constData, data, sizeof(data));
```

Listagem 2.1 – Código para variável na *memória constante*

A seguir será feita a indicação das principais características oferecidas pelos actuais *GPUs* da *NVIDIA*, como espaço de *memória partilhada* oferecida para os *blocos* ou número total de *threads* possíveis de serem executadas *activamente*, ou seja, todos aqueles que residem em cada *SM* disponível. Cada placa gráfica pode apenas ser compatível com algumas especificações oferecidas pelos *GPUs* da *NVIDIA*. Essas especificações estão compreendidas por entre quatro tipos, normalmente denominados por 1.0, 1.1, 1.2 e 1.3. Nesta dissertação foram apenas utilizadas placas com especificações de computação até 1.1.

2.4.4.6. Especificações de Computação (até 1.1)

- Número máximo de threads por bloco: 512
- Maior número de *blocos*: (65535 blocos, 65535 blocos, 1 bloco)
- Maior número de *threads*: (512 threads, 512 threads, 64 threads)
- Tamanho de um *warp*: 32 threads. (Nas versões actuais das placas gráficas é possível alterar este número, mas na dissertação essa possibilidade não é utilizada.)
- Número de registos para os *threads* por multiprocessador (SM): 8192
- Espaço total de memória partilhada por multiprocessador (SM): 16 Kb
- Número máximo de *blocos* activos por SM: 8
- Número máximo de *warps* activos por SM: 24
- Número máximo de *threads* activos por SM: 768
- Espaço total de memória constante por multiprocessador (SM): 64 Kb
- Espaço total de memória constante por multiprocessador (SM) para *cache*: 8 Kb

Aqui só é apresentada a informação relevante para a dissertação. Todas as especificações apresentadas são consideradas como especificações *por norma* para todas as placas gráficas da *NVIDIA*.

Nas próximas secções, toda esta informação é utilizada como base, porque toda a implementação foi optimizada de acordo com este padrão, tal como os resultados foram discutidos nessa base. No capítulo 3 será discutida a implementação e serão utilizados todos os conceitos aqui apresentados.

2.5. Algoritmos de Filtragem

O encaminhamento baseado no conteúdo, tal como usado em alguns sistemas editor/assinante, tem como ingrediente fundamental o algoritmo de filtragem. Uma possível abordagem com vista a melhoria de desempenho de um sistema desta natureza reside na sua paralelização, tirando partido do elevado poder de processamento dos processadores com arquitecturas baseadas em múltiplos *cores* que existem hoje em dia a preços competitivos.

Nesta secção, vão ser apresentados e discutidos os algoritmos de filtragem disponíveis na literatura que melhor se enquadram no problema apresentado. A intenção passa por entender os algoritmos e fazer uma avaliação prévia da sua facilidade de implementação em processadores com arquitectura de natureza paralela e mais especificamente em *GPUs* da última geração.

Numa primeira fase, os algoritmos de filtragem (sequenciais) existentes podem ser classificados nas seguintes categorias: árvores de decisão, diagramas binários de decisão; e algoritmos de contagem. Estas três abordagens são bastante diferentes entre si, com um impacto

considerável no modo como a sua paralelização pode ser abordada. Os seguintes parágrafos irão procurar fazer uma primeira descrição destas formas diferentes de tratar o problema.

As classes de algoritmos que seguidamente se descrevem em termos gerais partilham de alguns pressupostos. Em primeiro lugar, partilham do objectivo comum, ou seja, o processo de *matching* ou filtragem que consiste em determinar o subconjunto de subscrições que aceitam uma dada notificação ou evento. Para tal, as subscrições são vistas como restrições operando sobre os atributos dos eventos. Então, um evento acaba por ser simplesmente uma lista de pares (atributo, valor). Por sua vez, um filtro ou subscrição é uma lista de predicados operando sobre a estrutura linear do evento, formando uma cadeia de conjunções ou de disjunções, dependendo do caso.

As *Árvores de Decisão* [11] são mais conhecidas pela sua utilização em sistemas de apoio à decisão, onde os ramos indicam os possíveis caminhos a percorrer. Uma árvore de decisão representa a disjunção de conjunções de restrições nos valores de atributos, sendo que cada ramo é uma conjunção de condições. O conjunto de ramos na árvore é disjunto, sendo possível representar qualquer função lógica na árvore, como por exemplo "*x or y*" ou "*x and y*".

A árvore tem três tipos de nós diferentes, um nó de decisão (normalmente representado por um quadrado), um nó de hipótese (normalmente representado por um círculo) e um nó de terminação (normalmente representado por um triângulo). Cada nó de decisão contém um teste a um atributo, sendo cada ramo descendente um valor possível. Um nó de terminação é uma folha na árvore, sendo o caminho até ela caracterizado pelo conjunto de resultados obtidos pelos ramos.

Um Diagrama Binário de Decisão [12] tem como objectivo a representação de uma função booleana. É normalmente implementado num grafo acíclico, onde os nós de decisão têm apenas dois ramos como resultados possíveis, ou 1 ou 0, ou verdadeiro ou falso, respectivamente. Num Diagrama Binário de Decisão, existem, ainda, dois nós terminais com o valor 0 ou 1, indicando se é aceite ou não. Tem dois ramos distintos, um correspondente ao *high node* e outro ao *low node*. O *high node* corresponde ao ramo a seguir quando o predicado no nó corrente é aceite e o *low node* será o inverso. Uma subscrição será aceite se atingir o nó terminal 1.

Os algoritmos baseados na contagem são de natureza e implementação mais simples que os anteriores, na medida em que envolvem, numa primeira fase, o preenchimento de estruturas vectoriais relativamente simples, com base na avaliação independente dos predicados existentes no conjunto das subscrições. Seguida da fase em que é feita uma contagem para determinar quais as subscrições que têm todos os seus predicados satisfeitos.

2.5.1. Árvores de Decisão

Por entre os vários algoritmos de filtragem, com estruturas base como árvores, existem vários com o mesmo paradigma, focando-se nas subscrições, porque são os dados que se mantêm mais estáticos nos sistemas editor/assinante. Portanto, são criadas sempre estruturas de dados complexas, cujo custo é amortizado no tempo, pois são usadas para tratar vários eventos até haver uma actualização nas subscrições.

Um dos algoritmos de filtragem que se baseou em árvores de decisão foi [13]. O algoritmo é eficaz e, em certa medida, simples para a obtenção dos resultados da filtragem entre eventos e as subscrições efectuadas. A complexidade temporal oferecida por este algoritmo é sublinear com o número de subscrições e a complexidade espacial é linear.

As subscrições são consideradas como um conjunto de predicados onde cada predicado corresponde a um teste (que pode ser, por exemplo, de igualdade, de menor ou igual, etc., ou seja, testes booleanos) da forma <atributo, operador de comparação, valor>. Cada teste consiste numa fórmula a ser aplicada a um dos atributos, do evento recebido, e que poderá ter como resultado verdadeiro ou falso.

Quando entra um novo evento no sistema, os seus atributos são comparados e testados com os predicados obtidos das subscrições.

Neste algoritmo, é realizado um pré-processamento das subscrições, sendo criada uma estrutura auxiliar que torna o algoritmo mais rápido. A estrutura consiste numa árvore de decisão, onde cada nó é um operador de comparação de um predicado, ou seja, um teste. Os ramos são os vários resultados do predicado, portanto, cada caminho até uma folha corresponde a uma subscrição. Cada nível da árvore caracteriza um atributo das subscrições.

A árvore poderá ter alguns ramos com o valor *don't care*, significando que esse predicado não é importante na subscrição a ser testada.

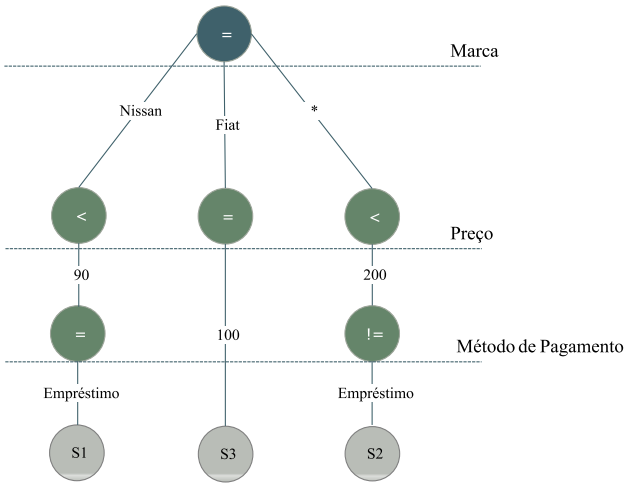


Figura 2.5 - Árvore de Decisão de 3 subscrições distintas. Estão representadas três novas subscrições acerca de vendas de carros, a S1 = "Marca = Nissan and Preço < 90 and Método de Pagamento = Empréstimo", S3 = "Marca = Fiat and Preço = 100" e por último S3 = "Preço < 200 and Método de Pagamento != Empréstimo". Existe um ramo "don´t care", na subscrição 2 não interessa a marca.

O algoritmo está dividido em duas partes. A primeira é o pré-processamento das subscrições, correspondente à criação da árvore dos predicados (das subscrições), e a segunda consiste, a cada entrada de um novo evento, na filtragem das subscrições no sistema.

O algoritmo de pré-processamento inicia uma árvore vazia, para a seguir analisar cada subscrição individualmente, ou seja, todos os predicados associados. Para cada subscrição são criados nós, que representam um operador de comparação, e cada ramo o valor no predicado, sendo analisado em qual nível da árvore será posto segundo o atributo do predicado, sendo posteriormente analisados aquando da inserção de outras subscrições para o caso de haverem predicados com operador igual, ou até mesmo valor, serem aproveitados, sendo que é possível vários nós e ramos serem partilhados.

O algoritmo de filtragem, que é processado quando entra um novo evento, percorre a árvore testando cada nó, o nó seguinte é escolhido consoante o resultado do teste anterior. As subscrições escolhidas são as folhas obtidas depois de a árvore ter sido percorrida. Neste algoritmo, pode ser utilizada uma busca em profundidade ou em largura para a obtenção das folhas correspondentes.

Quando as subscrições são um conjunto de predicados com testes de igualdade, é utilizada outra versão do algoritmo, especializada para estes casos, sendo provado que a complexidade temporal é sublinear segundo o número de subscrições. A diferença no algoritmo consiste na formação de níveis na árvore, cada nível é caracterizado por um atributo, sendo que desta vez o valor de cada nó é um atributo e cada ramo é o resultado do teste, ou seja, os valores da fórmula atómica dos predicados, por exemplo, na fórmula "nome = "Tiago", o valor seria então "Tiago" e o valor do nó seria "nome". Consoante o resultado do teste, o ramo está ligado a outro nó, sendo que mais uma vez esse caminho, até uma folha, caracteriza uma subscrição.

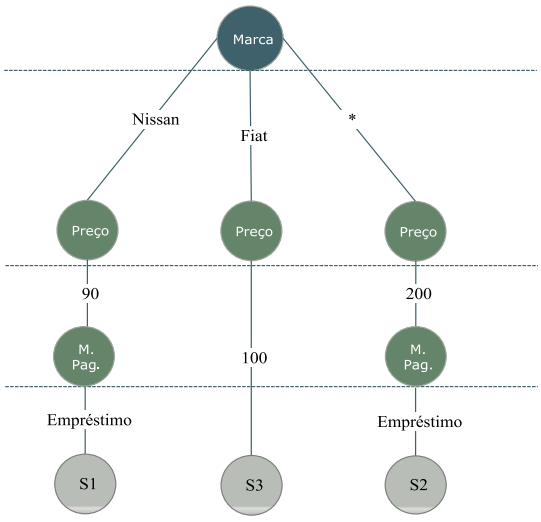


Figura 2.6 – Árvore de Decisão para subscrições de igualdade. Na árvore temos as subscrições: S1= "Marca = Nissan and Preço = 90 and Método de Pagamento = Empréstimo", etc…

Em relação à eventual paralelização, este algoritmo pode ser paralelizável de uma maneira imediata, que consistiria na divisão das subscrições pelos vários processadores, sendo que cada um formaria uma árvore distinta. No algoritmo de filtragem, na entrada no sistema de um novo evento, seria distribuído por cada um dos processadores. Processadores esses que iriam percorrer o evento consoante os seus atributos, devolvendo a(s) subscrição(ões) correspondentes aos testes positivos. Existe um ganho porque o processo de formação das árvores e o processo de filtragem seria concorrente, reduzindo o tempo, até porque com este algoritmo a complexidade temporal em muitos casos seria sublinear em relação as subscrições, o que deverá ocorrer com grande frequência dado que se espera que muitos predicados sejam partilhados entre as subscrições e, também, que a maior parte das subscrições dos utilizadores sejam igualdades, por exemplo, Noticias = Politica, Data = 22-04-1986, etc.

No que diz respeito à paralelização deste algoritmo em *GPUs*, é mais incerta e poderá mesmo não haver quaisquer ganhos, devido a dificuldades inerentes à arquitectura dos *GPUs*. Isto porque os processadores internos dos *GPUs* possuem memórias (locais) muito reduzidas (*partilhada* ou a *constante* por exemplo) e a complexidade para trabalhar com estruturas tão complexas, como árvores, é elevada. Como os *GPUs* funcionam maioritariamente por indexação de matrizes, não é claro se seria suficientemente vantajoso o esforço de exprimir o algoritmo segundo esse modelo. Portanto, o uso deste algoritmo nos *GPUs* está dificultado por esse facto, a elevada complexidade, tanto no algoritmo de filtragem, como na elevada complexidade na utilização da *API* para a implementação destas estruturas. Uma forma de aliviar o problema, poderia passar por não paralelizar o algoritmo de pré-processamento. Logo, as árvores poderiam ser obtidas no *host*, tornando assim mais fácil a criação das estruturas, mas continuaria a haver a dificuldade em transformar esse resultado numa estrutura fácil de utilizar nos *GPUs*.

As árvores de decisão são também utilizadas em [14], concretamente florestas, ou seja, conjuntos de árvores. Em concreto, a estrutura utilizada consiste numa *chained forest*, que mantém uma ordem parcial dos eventos e subscrições. Neste algoritmo, os eventos são *profiles* e as subscrições são *queries*. A noção de filtrar consiste em escolher quais as *queries* que fazem *match* com os *profiles*. Para tal, na inserção de uma *query*, é realizada uma procura de quais os *profiles* associados, e na inserção de um *profile* é também realizada uma procura. Existe uma floresta para todos os *profiles*, e outra para todas as *queries*. É portanto um algoritmo bastante complexo porque funciona com florestas, sendo também um algoritmo baseado em *DAG (Direct Acyclic Graph)*, que para manter as dependências são utilizados os filtros *poset* de *Siena* [15].

Para a sua paralelização, existem vários métodos, porque sendo um *workflow* pré-definido, cada processador podia processar e manter cada parte dependente, estando à espera de resultados, mas existe um desaproveitamento dos *CPUs* por estarem inactivos à espera dos

resultados, sendo que a melhor solução seria cada um ter um *DAG* pré-definido, sendo que cada um paralelamente processava os resultados.

Nos *GPUs*, no mínimo seria problemático implementar um algoritmo deste tipo, dada a relativa complexidade das diversas partes do algoritmo e a dificuldade de representação de estruturas como florestas dentro dos moldes algo rígidos dos *GPUs*.

2.5.2. Diagramas Binários de Decisão

Na aproximação do algoritmo [16], cada subscrição será representada num *Diagrama Binário de Decisão (BDD)*. Para isso, cada predicado, considerado uma fórmula booleana atómica, tem associado dois resultados possíveis, resultando na adição de um nó e dois arcos ao diagrama, correspondentes ao resultado verdadeiro e falso, respectivamente. Porém, se várias subscrições tiverem predicados iguais podem partilhar os mesmos nós, reduzindo a complexidade espacial do algoritmo. Dependendo do modo como são ligados os arcos entre os nós, é possível exprimir disjunções e conjunções de predicados.

Como cada subscrição, vista isoladamente, é também representada por um *BDD*, a partir deste podemos saber se a subscrição é aceite ou não, sendo que o último nó (folha) indica se a subscrição foi aceite ou não (tendo o valor *1* para aceite e *0* para rejeitado).

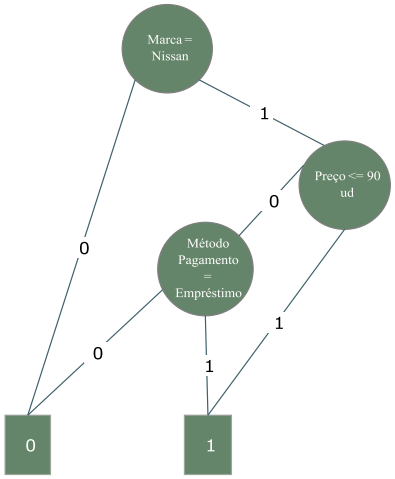


Figura 2.7 - Exemplo da representação de uma subscrição. A subscrição trata-se de um interesse num carro, de marca "*Nissan*", com o preço menor ou igual a 90 unidades, sendo que o método de pagamento pode ser com empréstimo, caso o valor do carro seja superior, ou seja, a expressão seria *Marca ="Nissan" and (Preço <= 90 or Método Pagamento = "Empréstimo")*. Se houver um evento com carros de marca "*Nissan*", com o valor superiores ou inferiores a 100 unidades, e com a possibilidade de empréstimo, esta subscrição seria aceite, porque, a marca é *Nissan*, iria para o nó do Preço, como o valor era superior o caminho escolhido seria o 0, sendo possível o método de pagamento por empréstimo, a subscrição atingia o estado final de aceitação. Mas, se o valor fosse superior a 90 e não tivesse o método de pagamento Empréstimo, então atingia o estado de não-aceitação, ou seja, o 0.

Mais uma vez, a paralelização deste algoritmo poderia passar pela ideia de dividir o problema pelos processadores disponíveis. Assim, as subscrições seriam repartidas e em cada um deles seriam formados os *BDDs* associados. A cada novo evento, seriam então percorridos os diagramas, separadamente em cada processador.

Pelas mesmas razões, referidas anteriormente, este tipo de solução parece um pouco difícil de suportar nos *GPUs*. Seria necessária a utilização de grafos acíclicos (árvores) e, como foi indicado anteriormente, o seu uso não é especialmente fácil. Possivelmente, o débito obtido não poderia ser competitivo quando comparado com a sua complexidade de implementação.

2.5.3. Vectores (Contagem)

Um algoritmo que explora um método de filtragem baseado na contagem está descrito em [17]. Como ponto de partida, temos que uma subscrição é considerada como sendo uma conjunção de predicados da forma:

(atributo, operador de comparação, valor)

Por exemplo:

(data, =, ”Novembro”)

Por seu turno, um evento, com nova informação será uma lista de elementos da forma:

(atributo, valor)

Ou seja:

(data, “13 de Novembro de 2008”)

Neste sistema, como esperado, as subscrições são aceites num dado evento quando todos os seus predicados são satisfeitos pelos valores dos atributos do evento. Ou seja, quando todos os atributos dos predicados da subscrição estão contidos no evento e satisfazem as operações de comparação. O algoritmo apresentado pelos autores é eficiente e rápido, aliado a uma metodologia bastante simples. O algoritmo correrá sempre que entra um novo evento no sistema.

O algoritmo trata apenas conjunções de predicados, sendo essa a única limitação do algoritmo a apontar. Mas, inicialmente, é associado um identificador único no sistema a cada subscrição. Esse identificador servirá para indicar, numa fase posterior, se essa subscrição foi ou não aceite. Depois, é criado um *vector de bits*, que tem como objectivo ser uma *máscara*, que indicará quais foram os predicados que, individualmente, foram aceites por um novo evento no sistema. Portanto, cada predicado também terá associado um identificador único no sistema, que refere um *bit* na *máscara* de predicados.

O tamanho desse vector irá variar consoante o número de subscrições no sistema e a distribuição dos predicados pelas mesmas. No caso de haver 100 subscrições com, em média, 10

predicados distintos, o tamanho do vector será portanto 10*100, 1000. Intuitivamente, cada posição corresponde a um predicado distinto no sistema. Com esta aproximação, as subscrições com predicados iguais partilharão o mesmo identificador, tornando assim a complexidade espacial da estrutura sublinear com o número total de predicados das subscrições.

A cada novo evento no sistema o comportamento do algoritmo será o seguinte: o valor de todas as posições do vector é colocado a 0, ou seja, não aceite. A seguir, é feita a avaliação de quais os predicados que satisfazem o evento. Isso é assinalado na máscara de bits colocando a 1 o bit correspondente a cada predicado que passa o teste. Por exemplo:

Predicado com identificador 30 da forma:

(“Notícia”,”⊆”,”exemplo”)

Evento da forma:

(“Notícia”, “Um exemplo está a ser demonstrado.”);
(“Data”, “13 de Fevereiro”);
(“Autor”,”João Araújo”);

O predicado seria filtrado e aceite neste contexto, porque, efectivamente, o atributo “Notícia” coincide em ambos e o valor no evento contém a palavra “exemplo”, como se pretendia no predicado. Sendo assim, na máscara, o valor do bit 30 será 1. Este procedimento é realizado para todos os predicados distintos no sistema, obtendo-se assim a máscara correspondente ao novo evento.

Na figura seguinte está representado o exemplo indicado.

Índice	...	26	27	28	29	30
Predicado	...	Marca, =, Fiat	Autor, =, João	Notícia, ⊆, exemplo 2	Data,=, 14 de Abril	Notícia, ⊆, exemplo
bit	...	0	1	0	0	1

Figura 2.8 – *Máscara de Bits*

Como se pode verificar, a cinzento temos a máscara de *bits*, indicando os predicados aceites, no caso os predicados com o identificador 27 e 30. Os passos apresentados anteriormente são sempre realizados quando um novo evento entra no sistema.

Mas, a organização das subscrições é distinta. A seguir, são apresentados os passos necessários a serem realizados por cada nova entrada de subscrições no sistema.

Basicamente, as subscrições são combinadas em a*gregados*. Um *agregado* é um conjunto de subscrições, com a particularidade de terem em comum o mesmo tamanho, ou seja,

o mesmo número de predicados. As subscrições são caracterizadas por um conjunto de predicados da forma:

(“atributo”, “operador”,”valor”)

Esse conjunto de predicados tem a particularidade, e limitação, de ser apenas uma conjunção como indicado anteriormente.

Cada *agregado* é representado por uma matriz rectangular de índices, onde as linhas correspondem aos predicados envolvidos e as colunas referem-se às subscrições pertencentes ao *agregado*. As matrizes são então de dimensão variável de *agregado* para *agregado*, isto é, *N***S*, onde *N* é o número total de predicados (igual para todas as subscrições nesse *agregado*), e *S* é o número de subscrições no *agregado*. Na figura seguinte é demonstrado um exemplo de como pode ser distribuído um *agregado*, neste caso com 8 subscrições de 4 predicados.

ID Subscrição	2	5	8	9	33	44	56	78
ID Predicado 1	2	6	4	13	78	4	5	3
ID Predicado 2	6	2	30	4	77	66	7	4
ID Predicado 3	9	33	27	22	4	77	4	45
ID Predicado 4	4	4	22	14	2	2	88	33

Figura 2.9 - *Agregado* para subscrições com tamanho 4

O exemplo é meramente ilustrativo. Num sistema real haverá certamente mais do que oito subscrições, podendo-se até situar-se na gama dos milhões. Mas, na primeira linha estão representados os identificadores das subscrições pertencentes ao *agregado* e nas linhas seguintes estão os índices da máscara de bits, permitindo assim indexar o resultado da avaliação individual dos predicados no seu todo. Então, o processamento de um *agregado* consistirá em contar os resultados positivos obtidos por cada subscrição. Como uma subscrição é uma conjunção de predicados, um evento só satisfará uma dada subscrição do *agregado* se *todos* os respectivos predicados tiverem o valor 1, ou em concreto, forem aceites. Existe, portanto, uma contagem implícita neste método. Alternativamente, um evento é rejeitado logo que se detecta um predicado não satisfeito, reduzindo assim o número de iterações necessárias para se saber se uma subscrição é ou não aceite.

Na figura seguinte é apresentado um exemplo do processo de contagem de uma subscrição, no caso, com identificador 8:

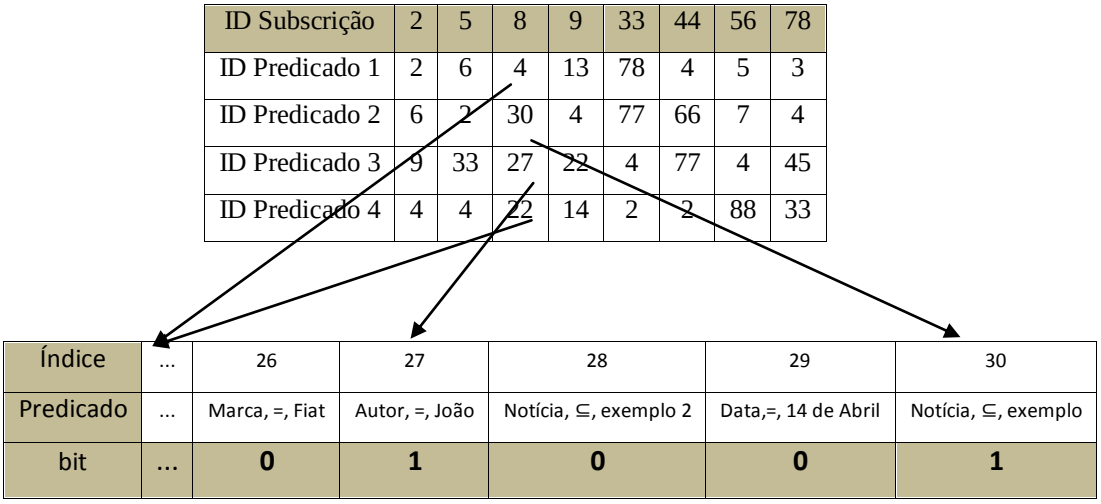


Figura 2.10 - Processo de contagem dos predicados das subscrições

A subscrição número 8, supondo que na posição 4 e 22 da máscara de bits está 1, é aceite. Porque todos os predicados foram aceites (estão a 1) pelo evento que entrou no sistema. Mas, por exemplo, a subscrição com identificador 44, o primeiro predicado é aceite, passando para o próximo predicado, supondo que o predicado 66 tem o valor 0 (ou seja, não foi aceite), a subscrição seria logo rejeitada, não sendo necessário verificar os próximos predicados. Como se pode verificar, é um algoritmo bastante simples.

A constatação que basta falhar um predicado para parar a contagem e dar uma subscrição como não satisfeita, leva a uma optimização suplementar, cujo objectivo é evitar contar ou processar conjuntos inteiros de *agregados*. Isso é possível se for encontrado um predicado especial, dito de *acesso*, com a particularidade de ser partilhado por todas as subscrições de um conjunto de *agregados*. É possível descartar vários *agregados* de uma só vez, sempre que o seu *predicado de acesso* não seja satisfeito pelo evento. Para tal, é feito um pré-processamento em que se criam listas de *agregados* que podem partilhar um mesmo *predicado de acesso*. Este processo não é fundamental para o algoritmo de contagem, mas permite ganhos substanciais, ao descartar muitas subscrições através da avaliação de uns poucos predicados críticos. Porém, neste processamento nem todos os predicados são eficazes e também é preciso, de tempos a tempos, refazer as listas de *agregados* em função da entrada e saída de subscrições no sistema. Por isso, é necessário estudar melhor as circunstâncias e saber se valerá a pena ou não aplicar esta optimização. Quando o número de subscrições for grande, a obtenção dos *predicados de acesso* pode ser um trabalho moroso, uma vez que será sempre necessário processar todas as subscrições. Esse processo no artigo não é bem explícito mas, para amortizar esse custo, quando entram novas subscrições, poder-se-ia guardar a informação dos seus predicados em estruturas de indexação, de modo a obter-se quais os predicados comuns ao

maior número de subscrições. Assim, reduz-se o trabalho, através da partilha, por entre todas as subscrições, dos predicados.

A seguir um exemplo é apresentado ilustrando a organização dos *agregados* através dos *predicados de acesso*.

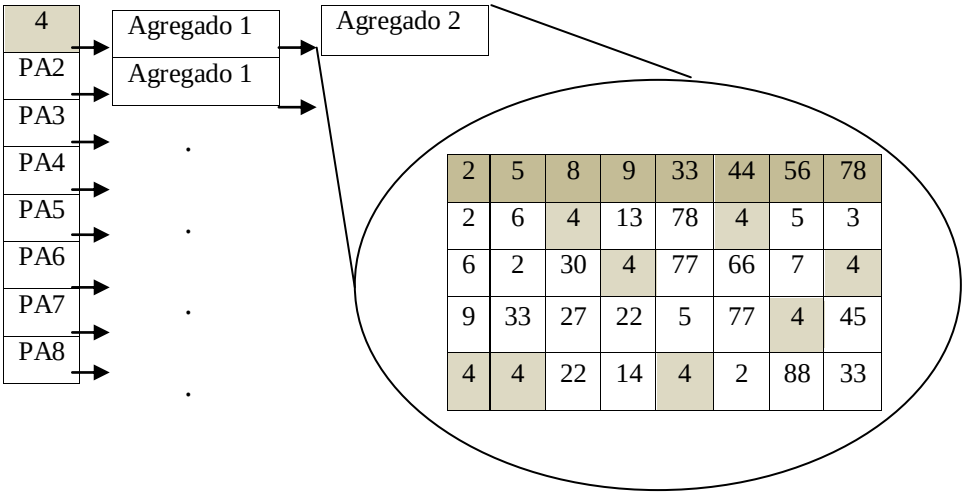


Figura 2.11 - Representação do Vector de *agregados*

No exemplo está como primeiro *predicado de acesso* o 4, ou seja, no caso, o agregado seria todo rejeitado se o predicado 4 fosse falso para um dado evento. Como se pode verificar, no segundo *agregado*, todas as subscrições têm o predicado 4 em comum. Como tal, está inerente que o predicado de acesso deve ser escolhido através dos predicados em comum por entre as subscrições, e se possível, que seja um predicado que falhe frequentemente no processo de filtragem, para assim haver uma maior probabilidade de um conjunto maior de subscrições falhar, havendo assim um ganho significativo na escolha de quais as subscrições aceites ou não.

2.5.3.1. Paralelização nos *GPUs*.

Uma das razões para a utilização deste algoritmo foi a simplicidade das estruturas utilizadas, tratando-se de estruturas lineares como vectores ou matrizes de 2 dimensões. Como neste algoritmo as operações fundamentais, e realizadas mais repetidamente, são as de contagem, torna o algoritmo a melhor escolha para implementação, porque pode-se retirar o melhor partido dos *GPUs*. Isso deve-se ao facto de apenas ser possível invocar um conjunto de *blocos* de *threads* por cada conjunto de código, ou seja, por cada *kernel*.

O grande conjunto de dados que se espera encontrar, em sistemas editor/assinante, também torna o algoritmo bastante bom, porque neste algoritmo os dados são organizados de uma forma bastante simples. Essa organização permite assim uma melhor divisão de trabalho por entre os *threads*, porque como se trata simplesmente de estruturas lineares (matrizes),

podem ser indexadas pelos índices únicos dos *threads*. A organização dos dados é de tal forma eficiente que a complexidade espacial da estrutura dos predicados é sublinear com o número de subscrições, já que os predicados são partilhados por entre as subscrições.

Ou seja, concluindo, devido a todas estas características, o algoritmo foi o escolhido para ser paralelizado nos *GPUs*. Como, em grande parte, o algoritmo baseia-se no preenchimento e processamento de vectores e matrizes de dimensões conhecidas, é parcialmente codificado no modelo de computação dos *GPUs*.

No capítulo *implementação* será realizada uma melhor analogia entre este algoritmo e a implementação que foi realizada nos *GPUs*, como também serão discutidos os resultados obtidos no capítulo *validação experimental*.

2.5.4. Outros (*XML*)

Há algoritmos recentes que utilizam directamente o *XML* e o *XPATH*, entre eles contam-se o *XFilter* [19] e o *YFilter* [20,21,22].

XFilter e *YFilter* foram propostos para facilitar o *matching* entre o *XPATH* e o *XML* e, para tal, ambas utilizam máquinas de estados finitas. No caso concreto do *XFilter*, cada expressão *XPATH* é primeiro transformada numa máquina de estados finita, e utilizando uma nova estrutura de indexação, permite que na entrada de um novo documento (como se fosse um novo evento), cada máquina de estados corra ao mesmo tempo, tratando várias expressões em paralelo. É utilizado o *parser SAX* [35], que a cada novo elemento no *XML*, faz o algoritmo processar as máquinas de estados; se alguma delas chegar a um estado de aceitação, significa que a expressão *XPATH* respectiva é aceite.

O *YFilter* foi proposto como uma evolução do *XFilter*, sendo que, em vez de cada expressão *XPATH* ter a sua máquina de estados, haveria no sistema um autómato finito não-determinista, onde as *tags* iguais nas expressões *XPATH* seriam partilhadas, reduzindo assim o espaço de ocupação das mesmas.

Em [9] é apresentada outra melhoria ao *XFilter* que consiste em substituir a estrutura de indexação por uma nova estrutura denominada *XTrie*. Através desta nova estrutura consegue-se reduzir o número de buscas desnecessárias em relação ao original. À mesma, o *parsing* dos ficheiros de *XML* e das expressões *XPATH* é realizado utilizando também o *parser SAX*.

Quanto à questão da paralelização, qualquer destes algoritmos o pode ser usando a técnica da divisão dos dados, ou seja as expressões *XPATH* a testar com um dado documento *XML*. Porém, a questão fundamental é que estes algoritmos destinam-se a fazer processamento de expressões *XPATH* e ficheiros *XML* genéricos e, por isso, são excessivamente complexos para o âmbito restrito desta dissertação. Na medida em que os eventos que vamos tratar são relativamente simples, consistindo em listas de *tags*, e o *overhead* associado à criação e

manutenção das estruturas de dados destes algoritmos dificilmente seria amortizado. Por maioria de razão, não faz muito sentido considerar qualquer um deles para implementação num *GPU*.

Na secção seguinte será feito um levantamento dos algoritmos anteriormente apresentados sob a forma de uma taxonomia. A taxonomia estará dividida em duas partes distintas, a primeira consiste na informação relevante dos algoritmos e a segunda parte caracteriza o tipo de estruturas utilizadas.

Na taxonomia, é apresentado mais um algoritmo [18], que se localizaria na classe de algoritmos de contagem. Não foi apresentado mais por extenso por motivos de espaço e porque as suas semelhanças com o [17] levaram a tal fim, sendo uma solução futura a ser explorada.

2.5.5. Taxonomia

A taxonomia seguinte apresentará de uma forma muito sucinta toda a informação relevante dos algoritmos estudados na secção anterior. A informação apresentada consistirá basicamente na sua descrição, o método de filtragem, tipo de subscrições e de eventos e por último, operações possíveis nos predicados. Esta informação torna-se fulcral para um melhor entendimento do algoritmo porque dá um panorama geral do seu funcionamento.

Na taxonomia seguinte será feita uma pequena reflexão da sua viabilidade na paralelização, quer em *CPUs* como em *GPUs*. Serão apresentadas também as complexidades temporais e espaciais de cada estrutura demonstradas pelos autores.

Nome Algoritmo	Descrição	Matching	Tipos de Subscrições	Tipos de Eventos	Tipos de predicados
Matching Events in a Content-based Subscription System[13]	É criada uma árvore para as subscrições. Os nós, vão ser os predicados, os ramos vão ser o resultado dos testes. O atributo é caracterizado por um nível na árvore. Como o que muda mais frequentemente são os eventos, as subscrições são mantidas em árvores para depois fazerem *matching* com cada evento novo.	Para um dado evento, as subscrições aceites são as folhas atingidas pelo algoritmo.	Conjunto de predicados da forma <atributo, operador de comparação, valor>	Conjunto de elementos da forma <atributo, valor>	Todos, ou seja, <, <=, =, !=, >= e >
Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems[17]	Cada subscrição tem um ID único. A cada evento, que entra no sistema, é criado uma máscara para todos os predicados, sendo que em cada posição é representado um predicado, onde 1 indica que fez *match* e 0 o inverso. De seguida, as subscrições são organizadas em *agregados*, caracterizados por conterem as subscrições com o mesmo número de predicados. Para cada grupo é criada uma matriz, cada posição da matriz corresponde ao ID do predicado na máscara.	Haverá *matching* quando a coluna correspondente á subscrição tiver todos valores a 1. Em cada posição, vai conter o identificador do predicado.	Conjunto de predicados da forma <atributo, operador de comparação, valor>	Da forma <atributo, valor>	Todos, ou seja, <, <=, =, !=, >= e >
Chained Forests for Fast Subsumption Matching[14]	Haverá uma árvore para as subscrições e outra para os eventos. Através de um evento, percorre-se a árvore de subscrições e obtém-se quais fazem *match*, e vice-versa para as subscrições, onde percorremos a floresta dos eventos.	A cada evento percorre-se a árvore de subscrições e obtém-se quais fazem *match*.	Consideradas como *queries*	Consideradas como *profiles*	Todos, ou seja, <, <=, =, !=, >= e >
Predicate Matching and Subscription Matching in Publish/Subscribe Systems[18]	Todos os predicados têm um *ID*, sendo indexados numa máscara. Os predicados do mesmo atributo e com o mesmo operador ficam numa lista ligada ordenada. As listas vão estar num dicionário com chave igual ao nome do atributo. Cada predicado terá um domínio associado, caracterizado por uma tabela bidimensional onde as linhas serão os operadores e as colunas os valores. O algoritmo está limitado a predicados com valores finitos.	Para o match das subscrições são criados um bit vector e um *hit vector*. No final é comparado esse vector com o *hit* vector, sendo devolvida essa subscrição se todos os predicados forem aceites.	Conjunto de predicados da forma <atributo, operador de comparação, valor>	Da forma <atributo, valor>	Todos, ou seja, <, <=, =, !=, >= e >
Efficient Filtering of XML Documents with XPath Expressions[9]	Cada subscrição é representada numa *árvore*, ou seja, para cada conjunto de *substrings*, do *XPATH* recebido. (possíveis combinações entre os atributos e "/"). Para quando entrar um *XML* de um evento, percorrer-se o *XML* e obter-se todos os nós.	Por cada documento *XML* é convertido para uma estrutura em árvore, percorrendo então a *Trie* de subscrições em busca de quais fazem *match*.	*XPATH*	*XML*	Todas que *XPATH* é compatível, ou seja, <, <=, =, !=, >= e >
Efficient Filtering in Publish-Subscribe Systems using Binary Decision Diagrams [16]	São utilizadas várias linguagens para *querys*, sendo utilizadas para submeter as subscrições. As quais são *SiSL*, *StSL* e *DeSL*. São utilizadas as estruturas *BDD* para a representação de todas as subscrições, os atributos de um evento são convertidos em booleanos, para serem comparados na *BDD*.	Para a obtenção das subscrições que fazem *matching*, converte-se o evento e percorre-se as *BDD*.	Fórmulas atómicas.	XML	Todas utilizadas nas fórmulas atómicas.

Tabela 2.3 - Informação dos Algoritmos

Estruturas de Dados	Paralelismo	Paralelismo nos GPUs	Complexidade Temporal	Complexidade Espacial
Árvores de decisão. [13]	Possibilidade de divisão de subscrições por diversas árvores correspondentes a um conjunto de subscrições, para quando entrar um evento fazer *matching* em paralelo.	Dificuldade com as limitações e complexidade do uso das memórias e das estruturas. Mas a solução passaria pela apresentada no paralelismo.	Linear com as subscrições, mas em casos de subscrições só com predicados de igualdade então será sublinear com as subscrições.	Linear com o número de subscrições
Vectores, 1D (bitmap e conjunto de Agregados) e 2D (Subscrições) [17]	Divisão dos *Agregados* por vários intervenientes, onde cada um executavam o algoritmo de *matching* para a obtenção das subscrições que fizeram *match*.	Estratégia de divisão do trabalho entre os vários processadores, tornando mais fácil definir intervalos de índices das estruturas de dados que irão estipular o trabalho a realizar por cada um dos processadores através dos identificadores dos *threads*.	Linear com o número de predicados	#Subscrições * Media(Predicados)/Subscrições
Árvores como florestas ligadas (*DoubleForest* ou *TripleForest)* [14]	Dividir cada fase do *workflow* por entre os vários processadores.	Muito difícil por razões de arquitectura, cada conjunto de árvores não seria fácil de implementar dado os moldes rígidos dos *GPUs*.	*O* (|Q| + |*P*|) Sendo Q = *Queries* P = Profiles	O (|P| · |Q|) Sendo Q = *Queries* P = Profiles
Vectores, 1D (máscara) e 2D (Predicados, onde colunas são o limite dos valores e as linhas são as operações de comparação) [18]	Dividir a lista de predicados dos vários atributos pelos processadores. Cada processador terá um dicionário para todos os atributos representados nesse processador para acesso na entrada de um novo evento.	Bom algoritmo para implementação, por serem estruturas vectoriais, mas existe um senão, a estrutura para a representação dos predicados tem um tamanho fixo. O *overhead* seria adicionado no processamento dos predicados para obtenção dos limites da estrutura. Haveria outro método como criar estruturas com tamanho pré-definido mas haveria a possibilidade de falsos negativos ou positivos.	Sublinear com as subscrições.	Sublinear com as subscrições.
Xtrie, ou árvores, onde cada subscrição terá a sua árvore. Listas, para as *substrings* [9]	Utilizando a técnica da divisão dos dados, ou seja as expressões *XPATH* a testar com um dado documento *XML*	Dificuldade nas estruturas e sua manutenção levam a que este algoritmo não seja a melhor escolha.	O(P . L . H . Lmax) *P* corresponde ao tamanho do maior caminho da raiz até a uma folha, *L* corresponde ao tamanho da maior lista ligada, *H* corresponde à maior altura de entre as árvores das *substrings*. Sendo esta complexidade correspondente a *tag* iniciais de um documento.	$\sum_{i=1}^{\|P\|} \|pi\|$ Sendo *P* o conjunto de *XPathExpressions* que estão indexadas e *pi* corresponde ao número de *substrings* na decomposição simples de *pi*.
BDD, Binary Decision Diagrams ou Diagramas Binários de Decisão, sendo representados em grafos acíclicos [16]	Por cada processador dividia-se os *BDDs* para uma mais rápida e eficiente filtragem dos eventos. A cada novo evento, seria enviado para cada processador para se utilizar o algoritmo de filtragem e obter-se as subscrições.	Estruturas demasiado complexas para serem implementadas na arquitectura dos *GPUs*. Devido também às limitações oferecidas pelos tipos de dados, também seria algo complicado implementar este algoritmo.	Sublinear com os predicados das subscrições.	Sublinear com os predicados das subscrições. (porque as *BDD* são o conjunto dos predicados das subscrições, sendo que quando existem iguais são reutilizados os nós, portanto, tornando-se sublinear.)

Tabela 2.4 - Estruturas e Paralelismo dos Algoritmos

Pode-se concluir que os focos fundamentais da dissertação vão incidir fundamentalmente no algoritmo estudado em [17], porque acaba por ser o escolhido para ser implementado nos *GPUs*. O critério para tal escolha residiu no facto de este algoritmo se basear em estruturas lineares e por ter as melhores hipóteses de poder ser codificado no modelo de computação. A própria estratégia de divisão do trabalho entre os vários processadores casou bem com esse modelo, pois torna-se fácil definir intervalos de índices das estruturas de dados que irão estipular o trabalho a realizar por cada um dos *blocos* de *threads* através dos identificadores únicos existentes e já mencionados. O processamento dos identificadores dos elementos (como os atributos, predicados, etc…) é todo realizado no *host*, para facilitar o processo, sendo que o cálculo da máscara e a obtenção das subscrições aceites é totalmente implementada e executada nos *GPUs*.

3. Implementação

Na forma fundamental dos sistemas editor/assinante, os assinantes submetem uma ou mais subscrições para obterem apenas os eventos do seu interesse. As subscrições são tipicamente conjuntos de predicados. Os predicados representam todos os *filtros* que os assinantes pretendem que sejam satisfeitos.

No algoritmo seleccionado [17], os autores descrevem uma implementação de um sistema de gestão de subscrições. As subscrições são consideradas *sempre* como uma *conjunção* de predicados, ou seja, a sua representação é algo como:

$$S_i = P_1 \wedge P_2 \wedge P_3 \ldots \wedge P_n$$

Os predicados têm a forma:

$$P_i = (Atributo, Operador, Valor)$$

O *Atributo* é o único elemento que os predicados podem partilhar com os eventos. Os eventos são considerados como um conjunto de elementos da forma:

$$E_i = (Atributo, Valor)$$

Portanto, no processo de filtragem basta comparar os elementos *Atributo,* entre o evento e o predicado, para saber se há possibilidade ou não da aceitação. Se sim, verifica-se os valores, segundo o operador de comparação. Se for satisfeito, o predicado é aceite.

Tanto as subscrições como os predicados no sistema têm um identificador único. Os predicados são indexados, através do seu identificador único, numa estrutura de *bits* com o nome *máscara de bits.* Esta estrutura é actualizada sempre que entra um novo evento porque é nela que vai estar toda a informação de quais os predicados aceites.

As subscrições no algoritmo são organizadas e distribuídas por *agregados*. Cada *agregado* tem a característica de conter apenas as subscrições com o mesmo número de predicados, ou seja, com o mesmo *tamanho*. Existindo *n tamanhos* diferentes por entre todas as subscrições, necessariamente terão de existir *n agregados* distintos para a sua distribuição. Como as subscrições são um conjunto de predicados, essas são representadas como um conjunto de identificadores desses predicados, que indexam a *máscara de bits.* Sucintamente, um *agregado* é uma matriz de identificadores de predicados. Uma coluna é uma subscrição e as linhas associadas são os valores dos predicados.

Basicamente, o funcionamento do algoritmo está dividido em duas fases distintas, a obtenção da *máscara* e o processo de filtragem das subscrições.

O processo da obtenção da *máscara* consiste no teste de todos os predicados para um dado evento. Sempre que o resultado devolvido do teste é igual a rejeitado, o valor colocado na máscara é 0, sendo o valor de aceite igual a 1.

O processo de filtragem tem o conceito inerente de contagem. Esse processo resume-se apenas à contagem dos predicados aceites por cada subscrição. Como uma subscrição é um conjunto de identificadores dos predicados, que indexam a *máscara*, basta contar os valores a 1 (aceites) nessa estrutura. Assim, uma subscrição é apenas aceite se *todos* os predicados são satisfeitos.

Uma optimização proposta pelos autores foi a possibilidade de organização dos *agregados* segundo um *predicado de acesso*. Esse predicado ditava qual o resultado da filtragem do *agregado*, ou seja, bastava esse falhar para todo o conjunto de subscrições ser descartado. É um processo eficiente porque, possivelmente, um grande conjunto de subscrições falha o processo. Mas, para essa eficiência ser obtida o predicado escolhido teria que ter uma grande probabilidade de falhar porque senão de nada valia esta optimização.

Na implementação realizada, face ao algoritmo original, as principais características foram mantidas. Entre as quais a representação dos predicados e subscrições. As subscrições também são um conjunto de identificadores dos predicados e são também utilizados para referenciar a *máscara*. A *máscara* também não sofreu qualquer alteração.

As grandes alterações realizadas foram a nível da distribuição das subscrições. Na implementação, os *agregados* têm um tamanho pré-definido, ou seja, de antemão apenas podem manter um número pré-determinado de subscrições. *A priori*, o número de *agregados* no sistema também é fixo, tal como o *tamanho* das subscrições aceites em cada. Foram definidos como *tamanhos* máximos permitidos nos agregados, para as subscrições, valores de potências de dois para facilitar o processo de distribuição na implementação. Assim torna-se mais acessível a gestão dos *agregados* porque à partida o sistema já sabe qual o tamanho desses.

As duas fases distintas do algoritmo foram divididas pelas duas unidades de processamento disponíveis, ou seja, pelo *host (CPU)* e pelo *device (GPU)*. O papel do *host* está mais vocacionado para a manutenção das estruturas de dados, porque nele podem ser utilizadas, de uma forma mais acessível, estruturas de indexação, como dicionários. O *device* terá um papel mais activo na obtenção dos resultados, ou seja, nos processos de cálculo da *máscara* e de filtragem das subscrições através de *kernels*.

O *host* obtém toda a informação dos elementos e distribui-a pelas estruturas definidas, como por exemplo, os *agregados*. Todo esse processo é realizado no *host* porque a manutenção dos elementos no sistema é fundamental para a coerência entre todos. O *host* suporta a utilização de estruturas mais complexas, o que possibilita a gestão mais fácil da informação.

Todas as estruturas lineares obtidas são transferidas para o *device* de modo a que este possa obter todos os valores necessários para a filtragem. Essa transferência é realizada pelo *host* e por isso é considerada a unidade que gere qual o trabalho que o *device* deve realizar. Note-se que quando o *device* está a realizar uma tarefa e não pode realizar outra, o *host* pode ter um papel activo e cooperar com o *device* processando o trabalho que este não pode realizar.

Há a necessidade de tornar o sistema dinâmico e com grande escalabilidade portanto tem que se garantir a entrada e saídas de subscrições do sistema. Esse processo é todo realizado no lado do *host* e apenas são transferidas para o *device* as posições que são necessárias para remover ou então as subscrições que têm de ser adicionadas. Esta funcionalidade, que não foi discutida no algoritmo original, foi estudada nesta dissertação.

No *device* foram implementados todos os *kernel* necessários para as duas fases do algoritmo. A obtenção da *máscara* é realizada com recurso a apenas um *kernel*, que recebe toda a informação do *host.* Essa informação consiste nos predicados e no evento introduzidos no sistema. O processo de filtragem é realizado depois da obtenção da *máscara* e também é processado com recurso a apenas um *kernel*, executado 1 vez por cada um dos *agregados*.

Em relação às optimizações propostas pelos autores do algoritmo, não foi considerada a implementação dos *predicados de acesso* por se revelar um processo bastante complexo para o tempo disponível para a execução do trabalho experimental desta dissertação. Mas, em contrapartida, foi implementada a possibilidade de o sistema suportar subscrições com *disjunções* e *negações* de predicados. Outra optimização foi a possibilidade de se filtrar mais de que um evento de cada vez. Essa optimização justifica-se porque tipicamente em sistemas editor/assinante de larga escala o fluxo de entrada de eventos espera-se bastante elevado. A possibilidade de filtrar mais do que 1 de cada vez foi também motivada por factores relacionados com a exploração da granularidade da paralelização.

Seguidamente, vão ser apresentados todos os pontos focados até aqui, ou seja, como foram implementadas todas as características do algoritmo, e serão feitas as analogias, com o algoritmo escolhido, necessárias para uma melhor compreensão da implementação.

Inicialmente, é apresentada uma implementação mais próxima do algoritmo. A sua explicação é detalhada, tanto na parte do *host* como no *device*. Mas, com um maior conhecimento adquirido, e com o surgimento de maiores oportunidades ao longo do seu desenvolvimento, algumas alterações foram realizadas no código. Essas alterações permitiram obter melhores resultados. Tais alterações são apresentadas na secção *optimizações*.

3.1. *CPU (Host)*

No *host*, são geridos quatro elementos principais: os atributos, os predicados, os eventos e os *agregados* (subscrições).

Os predicados e eventos são elementos necessários para o cálculo da *máscara*. Os *agregados* são os principais elementos para o processo de contagem. Estes elementos são todos representados em estruturas lineares.

Além dos tipos nativos oferecidos no *ANSI-C*, a *API* do *CUDA* oferece várias extensões desses sendo tratados como novos tipos nativos. Vários foram utilizados para facilitar a implementação, entre eles, os tipos: *int3*, *int4*, *float4* e *int2*, que já foram mencionados na secção *arquitectura (GPU)*.

Seguidamente, será feita uma explicação de como estão organizados os elementos e as suas estruturas. A implementação que vai ser apresentada seguidamente apenas filtra um evento de cada vez.

3.1.1. Predicados

Os predicados, como indicado, têm a forma:

$$P_i = (Atributo, Operador, Valor)$$

Os predicados no sistema têm um identificador único, obtido através de uma variável global que indica o *número total de predicados* existentes.

A sua representação no *host* foi obtida com recurso a várias estruturas. De uma forma concisa, o número das estruturas utilizadas são quatro. Um vector de *int3*, um vector de *float4* e dois dicionários.

O vector de *int3* representa o *conjunto de predicados*. Em cada posição deste vector está toda a informação de um predicado. Um dos objectivos do identificador único de cada predicado consiste na indexação desta estrutura para a obtenção da sua informação.

Um *int3* é:

Predicado.x = IDAtr
Predicado.y = Oper
Predicado.z = Índice na estrutura dos valores

IDAtr representa o identificador único do atributo no sistema. Esses identificadores são utilizados para evitar a redundância dos atributos por entre os vários predicados e eventos. Na secção *atributos* é apresentada uma explicação sucinta de como esta questão é realizada.

O elemento *Oper* é um inteiro que indica qual o operador. Por exemplo, o valor 0 indica que é uma igualdade. Na secção *operadores* é apresentada uma tabela com todos os operadores suportados no sistema.

O terceiro inteiro indica o índice para uma posição no vector dos *valores dos predicados.* Esse vector corresponde ao vector de *float4*.

O vector de *valores de predicados* tem todos os elementos *valor* dos predicados. A opção de representar os valores em *float4* deve-se à possibilidade de a implementação suportar valores reais, como por exemplo, percentagens ou num exemplo mais concreto, preços de objectos.

Foi tomada, neste passo, uma das decisões mais importantes, o tratamento das *strings*. Como no *device* a dificuldade de utilização e de manipulação de *strings* é bastante elevada, uma forma directa e simples de as suportar passou por converte-las num valor de dimensão constante. Como o tamanho das *strings* é bastante variável, para facilitar essa conversão, optou-se por se utilizar uma função de *hash* bastante conhecida, *SHA-256*. Assim, é garantido que qualquer que seja o tamanho da *string*, o seu tamanho será sempre de 256 *bits*. De momento assume-se que não há colisões, apesar de esse condicionante acrescentar apenas *falsos positivos* no resultado.

De notar que as estruturas indicadas até agora têm o objectivo de serem utilizadas para a obtenção da *máscara*, que, como foi indicado, é também indexada pelos identificadores de cada predicado.

As estruturas seguintes têm como objectivo a manutenção dos predicados de uma forma coerente, por entre todas as subscrições.

Um dos dicionários tem o intuito de guardar, para cada novo valor distinto dos predicados, a posição desse no vector de *valores de predicados.* Assim é garantida a sua partilha. A complexidade espacial da estrutura dos *valores de predicados* consequentemente é sublinear com o número de valores totais que entram no sistema porque apenas são guardados os valores distintos. O dicionário chama-se *dicionário de valores*.

Por último, o objectivo do outro dicionário é bastante simples. Como há grande probabilidade de existirem predicados iguais por todas as subscrições, a necessidade de evitar redundância torna-se crítica. Por exemplo, numa página de desporto e supondo que existem muitas subscrições com o predicado igual a:

Clube = CA

A chave do dicionário é a *string*, isto é, no exemplo seria "Clube=CA" (sem espaços), e o valor do dicionário é o identificador global do predicado, ou seja, o índice na estrutura do *conjunto de predicados.* Este dicionário é denominado *dicionário dos predicados.* Sempre que entra uma nova subscrição, que contém o predicado exemplo, e o sistema já tem esse valor, o

identificador único do predicado é obtido e é associado à subscrição. O processo é repetido para todas as subscrições que partilham o mesmo predicado.

Nas seguintes subsecções são apresentadas duas informações importantes, como são obtidos os identificadores únicos dos atributos e qual é a representação dos operadores no sistema, que é feita com recurso a inteiros. Por último, a secção *obtenção de predicados* exemplifica todo o processo realizado na entrada de um novo predicado no sistema.

3.1.1.1. Atributos

No sistema, os atributos têm um identificador único. Esse identificador é dado através de uma variável que indica quantos atributos já existem no sistema. Como os atributos são os únicos elementos em comum por entre os predicados das subscrições e os eventos, a garantia que esses não ocupam mais espaço que o necessário (redundância) e que os identificadores únicos no sistema são partilhados tem de ser obtida. Essa garantia é atingida através de um dicionário global no sistema.

A chave é o atributo e o valor o seu identificador. Na figura seguinte está representado um possível exemplo do dicionário indicado.

Chave	Valor
Marca	0
Modelo	1
Ano	2
Preço	3
...	...
...	...

Figura 3.1 – Dicionário dos Atributos

Estas chaves são partilhadas por entre os predicados e também por entre os eventos. Os valores são inteiros, obtidos na leitura do atributo.

3.1.1.2. Operadores

Neste trabalho foram considerados sete operadores de dois tipos, enumerados na tabela seguinte. O que distingue cada tipo de operador são os tipos de dados em que incidem, ou seja, se é utilizado para valores numéricos ou conjunto de valores alfanuméricos. De salientar que o último operador na tabela (operador lógico *contém*) pode operar sobre uma lista de valores.

Operador	Valor do Operador	Tipo de dados
=	0	Numéricos
>	1	Numéricos
<	2	Numéricos
!=	3	Numéricos
<=	4	Numéricos
>=	5	Numéricos
⊆	6	Strings

Tabela 3.1 – Operadores suportados

3.1.1.3. Obtenção de Predicados

A título de exemplo, a seguir é feita uma apresentação do processo de introdução de um predicado no sistema. Todos os passos consistem na obtenção dos inteiros necessários para a representação do *int3* de um predicado.

O predicado teste é:

Marca = Fiat

O primeiro passo passa pela verificação, através do *dicionário dos predicados*, se já existe ou não o predicado no sistema. Através da chave "*Marca=Fiat*" verifica-se se já existe essa chave associada. Supondo que não, introduz-se o predicado no *dicionário dos predicados*.

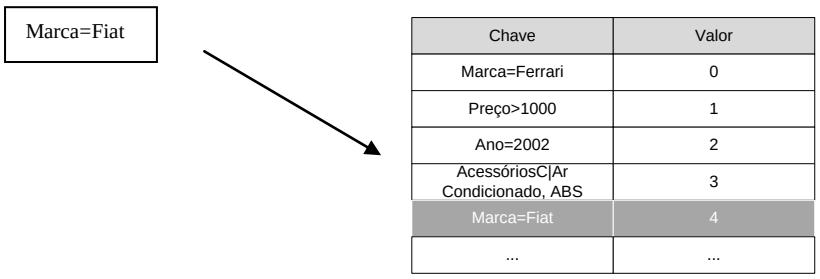


Figura 3.2 – Adição do novo predicado no dicionário

O segundo passo é a verificação da existência do atributo "*Marca*":

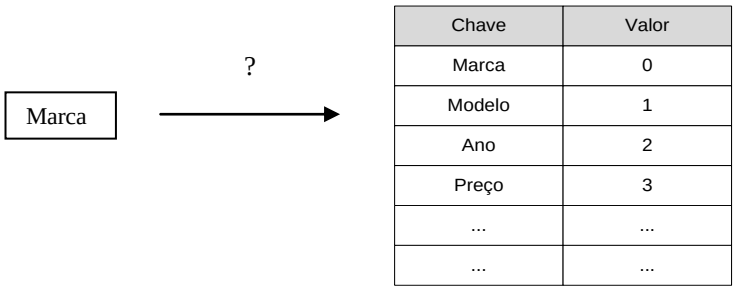


Figura 3.3 – Verificação da existência do atributo

Como existe, o valor X do *int3* será 0. A partir da tabela dos operadores retira-se facilmente qual o valor do operador. Portanto, o valor Y é 0.

Depois, é verificado se existe o valor do predicado no *dicionário de valores*. Se não existir e se o valor é uma *string* essa tem de ser convertida para os dois *float4*. Se for numérico, o valor é colocado na posição X do primeiro *float4*. No caso de ser uma *string*, converte-se esta para um *hash* de 256 *bits*.

Como não existe, obtém-se o número de valores distintos no sistema e adiciona-se no dicionário a *string* "*Fiat*" com o número 10. Ou seja:

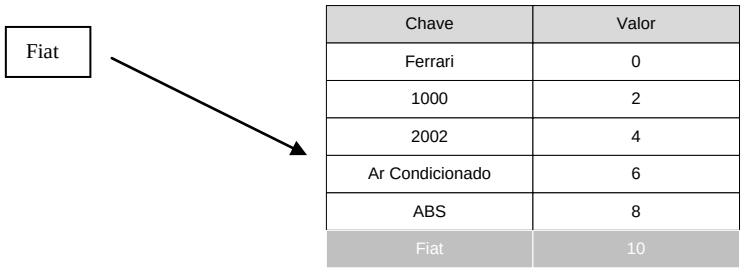


Figura 3.4 - Adição do novo valor "*Fiat*" no dicionário de valores

Os valores do dicionário são pares porque, como indicado anteriormente, um *valor* no vector de *valores de predicados* é representado em dois *float4*.

Finalmente, a representação do predicado num *int3* seria:

Predicado.x = 0
Predicado.y = 0
Predicado.z = 10

De uma forma esquemática, o funcionamento das estruturas é o seguinte:

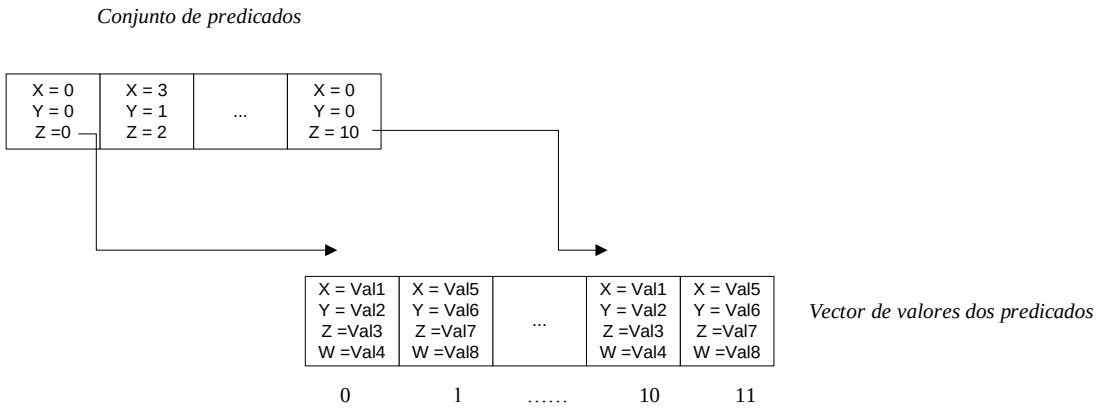


Figura 3.5 - Associação entre os vectores conjunto de predicados e os valores dos predicados.

Na figura 3.5 estão representadas todas as referências e ligações existentes entre as estruturas dos predicados. As ligações representadas fazem parte do primeiro e do último predicado no vector. Cada *valor* de um predicado é representado em duas posições no vector de *valores dos predicados*, porque são dois *float4*, e o valor Z do *int3* de um predicado é o índice para essa estrutura.

Todas as estruturas lineares apresentadas são transferidas, numa fase posterior, para o *device*, e são mantidas em ambas as memórias (*host* e *device*) de uma forma persistente, para o seu reaproveitamento nas operações de filtragem seguintes. Em ambos os casos, a adição e remoção de predicados é determinística.

3.1.2. Subscrições (*Agregados*)

As subscrições são um conjunto de predicados. O algoritmo estudado apenas foi desenvolvido para *conjunções* de predicados. Ou seja, considera-se uma subscrição como:

$$S_i = P_1 \wedge P_2 \wedge P_3 \ldots \wedge P_n$$

O tratamento das subscrições é linear. Sempre que entra uma nova subscrição, ela é decomposta. Como as subscrições são *conjunções* de predicados, esses são tratados de forma individual. Assim, uma subscrição é apenas um vector de inteiros, representativos dos identificadores dos predicados.

Uma das alterações na implementação, em relação ao algoritmo de referência, residiu na distribuição das subscrições. No algoritmo original, caso houvesse *n* subscrições com tamanhos diferentes, haveria *n agregados* distintos. Na implementação desenvolvida, o tamanho *requerido* para uma subscrição ser inserida nos agregados é pré-definido à partida. Significando

que o sistema fixa *a priori* o número de *agregados*, que também tem a capacidade máxima definida. Caso estes valores sejam insuficientes, delega-se no *host* o tratamento das subscrições excedentes ou com demasiados predicados. Concretamente, os *tamanhos* máximos para cada subscrição ser inserida num *agregado* são valores de potências de dois. Por exemplo, no caso de existirem quatro *agregados* distintos, os tamanhos para cada *agregado* serão: 2, 4, 8 e 16.

As subscrições com as seguintes características são aceites em cada *agregado*: tamanho entre 1 e 2 no *primeiro*, 3 e 4 no *segundo*, 5 e 8 no *terceiro* e no *quarto* 9 e 16.

Na figura seguinte está representada essa distribuição:

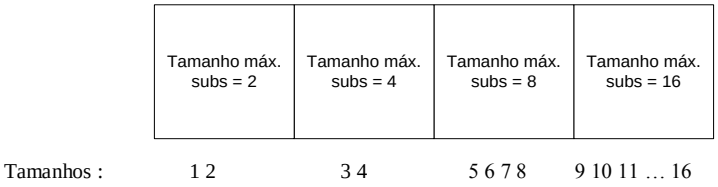


Figura 3.6 - Nova organização dos *agregados*

Estas restrições visam simplificar a gestão da memória no *GPU* e reduzir o número de *kernel* necessários para o processo de filtragem. Assim, a distribuição das subscrições tornou-se mais uniforme, tornando a gestão das estruturas mais simples porque não existe necessidade, à partida, de ter várias colunas numa matriz mas apenas umas quantas. As linhas da coluna, ou mais em concreto as subscrições, têm a mesma organização que o indicado no algoritmo escolhido. As subscrições que não tiverem como tamanho o número máximo de predicados possíveis no *agregado* terão o resto dos predicados preenchidos com um índice da *máscara* correspondente ao predicado "*true*", funcionando como um *elemento neutro* na filtragem. Na filtragem da subscrição, o *kernel* valida esses *bits*, não alterando o resultado final.

A organização das subscrições, no segundo *agregado* (subscrições com tamanho entre 3 e 4) podia ser algo como:

ID Sub	ID Sub	ID Sub	ID Sub
9	8	4	3
5	0	10	33
7	1	14	25
3	5	*n*	*n*

Figura 3.7 - Um *agregado* com subscrições de *tamanhos* até 4

O valor *n* refere-se ao índice da *máscara* correspondente ao predicado "*true*". O tratamento do resto das subscrições é análogo, mudando apenas o *agregado* onde estão inseridas. O número de subscrições que cada *agregado* pode receber é parametrizado no arranque do sistema. Estes parâmetros teriam que ser afinados de acordo com a distribuição esperada das subscrições e dos seus predicados.

Fazendo a analogia com o algoritmo, na figura seguinte está representada a organização das subscrições nos *agregados:*

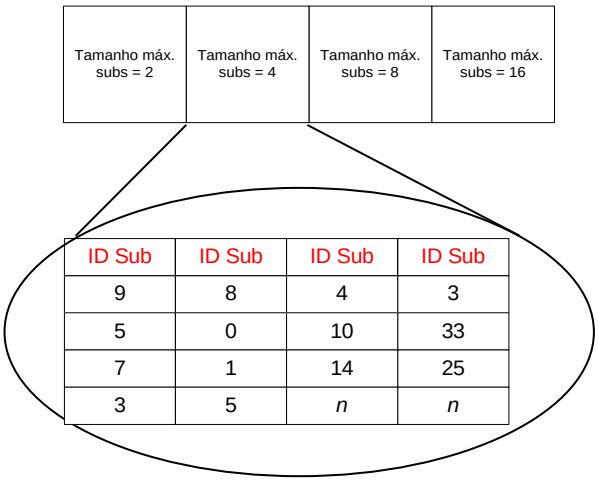


Figura 3.8 - Distribuição dos *Agregados*

Como se verifica, a organização é, de um modo geral, muito semelhante à apresentada no algoritmo. Apenas se deduz que o número de subscrições pode ser maior em cada *agregado* do que no algoritmo. Mas, na dissertação, esse custo é amortizado pela facilidade de implementação das estruturas e pela mais rápida distribuição das subscrições por cada *agregado*.

Em termos de estruturas, de forma sucinta, os *agregados* são vectores de inteiros de duas dimensões, ou seja, uma matriz de inteiros. O número de linhas da matriz varia consoante o número máximo de subscrições por *agregado*. Em relação às colunas, com *n agregados*, o vector teria *n + 1* colunas. A razão da coluna adicional baseia-se na sua utilização como estrutura de manutenção da informação. Essa informação indica apenas o número de subscrições já inseridas em cada *agregado*.

A estrutura seria algo como esta representação:

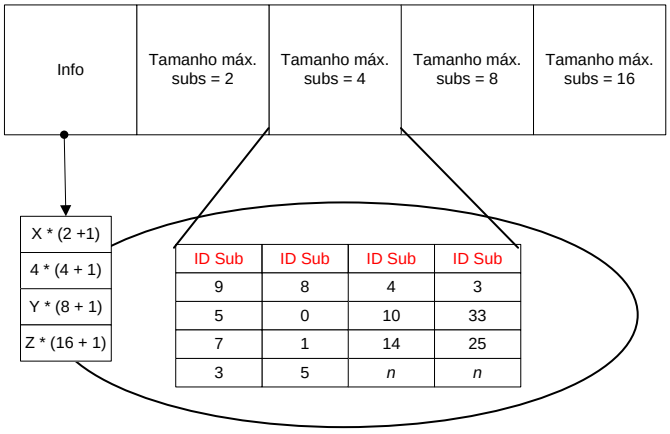


Figura 3.9 - Organização das subscrições no sistema

Na figura 3.9 está representado o vector com a informação dos *agregados*. A informação é utilizada para que, aquando do processo de introdução de novas subscrições no sistema, a obtenção de posições livres no *agregado* seja mais rápida (caso haja espaço no *agregado*).

Os valores X, Y e Z na figura são apenas indicativos do número de subscrições por cada *agregado*. No segundo *agregado* o valor é 4, como representado na matriz da figura. Facilmente se deduz a próxima posição a adicionar. Como o *agregado* tem quatro subscrições cada uma com tamanho quatro e uma posição extra para o seu identificador, a próxima posição livre será a 4 * (4+1), ou seja, a 20.

O conceito de matriz para um *agregado* é apenas uma representação *abstracta* da estrutura, porque no código efectivamente é representada num vector unidimensional de inteiros. O acesso aos valores é feito como se de uma matriz se tratasse, ou seja, através da fórmula:

*Coluna * (Tamanho da Coluna) + Linha*

Para o sistema poder vir a ser considerado robusto, a necessidade de torná-lo dinâmico é fundamental. Como tal, foram implementados métodos que possibilitam a introdução e remoção dinâmica das subscrições do sistema. Tal procedimento não é discutido no algoritmo original e como tal, a sua implementação foi realizada de raiz.

A manutenção das subscrições removidas ou adicionadas é realizada no lado do *host,* porque a utilização de estruturas mais complexas, como dicionários, é mais simples e facilita bastante este processo. Para a comunicação das subscrições alteradas entre o *host* e o *device* foi necessária a definição de períodos de comunicação. Essa comunicação consistia em transferir um determinado número de subscrições ou para remoção ou para adição. Se a comunicação for frequente, o custo é bastante elevado porque os acessos à *memória global* do dispositivo são tipicamente de elevado custo. Então, os períodos de comunicação têm de ser com menor frequência. Esses podem ser de *x* em *x* subscrições novas ou então num dado intervalo de tempo. Neste trabalho, esse período foi definido como um número *x* de subscrições removidas/adicionadas. Foram efectuados alguns testes para se obter o impacto dessa comunicação, que serão apresentados no capítulo *validação experimental.*

De uma forma geral, a sua implementação é a seguinte: existe um dicionário, de nome *subscrições,* que recebe como chave o identificador da subscrição, tendo como valor um *int3*. Esse valor tem como objectivo indicar a localização da subscrição, funcionando como um índice na matriz representativa dos *agregados*.

A informação desse *int3* é:

Subscrição.X = Identificador da Subscrição
Subscrição.Y = Número do agregado onde reside
Subscrição.Z = Posição no agregado onde reside

O primeiro valor tem um objectivo bastante importante na fase da remoção e adição, explicado nas secções seguintes. O segundo valor indica em qual dos *agregados* a subscrição se localiza e o terceiro valor indica em que posição se localiza nele. Esta estrutura é utilizada em ambos os casos, na adição e remoção de subscrições.

Seguidamente, duas secções serão apresentadas, uma para a adição e outra para a remoção das subscrições.

3.1.2.1. Remoção de Subscrições

Para a remoção de uma subscrição basta obter um dado, o identificador único. Esse identificador, numa fase posterior, será associado a outra subscrição nova no sistema. Mas, para que tal aconteça, a subscrição tem de ser removida. A remoção passa apenas por acrescentar um novo elemento num novo dicionário, chamado *subscrições removidas.* Nele, são acrescentados como chave o identificador da subscrição removida e como valor o mesmo valor *int3* que o dicionário *subscrições* contém. Esse será utilizado para indicar, numa fase posterior, quais as posições livres nos *agregados.* É nesta fase que o valor *X* do valor *int3* ganha relevo, nele está indicado qual será o futuro identificador de uma nova subscrição.

A fase de remoção pode passar por *remover* apenas uma subscrição da memória do *device* ou então um conjunto de subscrições. Na secção de resultados analisa-se esta questão, sendo apresentada uma conclusão para qual será o impacto do tamanho do conjunto de subscrições diferentes para remoção. Para um determinado número de subscrições removidas, o *host comunica* ao *device* quais as removidas através da indicação das posições no *agregado*. As posições são inseridas num vector de *int2*.

Esse vector de *int2* é guardado num vector de duas dimensões. Cada coluna representa um *agregado* e cada linha é um valor do vector de *int2*. O número de colunas desta matriz é igual ao número de *agregados* no sistema. Esta matriz é *transferida* numa fase posterior para o *device*.

O trabalho do *kernel* apenas consiste na mudança da referência do primeiro predicado da subscrição removida. A referência passa a ser a *m.* Na *máscara* o valor nessa posição é o 0, que corresponde ao *elemento absorvente* da conjunção ou predicado "false". Assim, de imediato

falha o processo de filtragem da subscrição. A implementação do *kernel* é apresentada na secção do *GPU (Device)*.

3.1.2.2. Adição de Subscrições

Na entrada de uma subscrição, depois da análise da sua estrutura, existe naturalmente a necessidade de a adicionar a um *agregado*. Mas, como o número máximo de subscrições por *agregado* é definido à partida, existe alguma probabilidade de não existir espaço para a sua adição. Efectivamente, só será adicionada quando outra subscrição é removida, obtendo assim uma *vaga n*o *agregado* a que pertence. Essa *vaga* é obtida através do dicionário indicado anteriormente, o dicionário *subscrições removidas.*

Mas, pode haver uma boa probabilidade de não ter esse lugar. Nesse caso, a subscrição é colocada noutro vector de duas dimensões de inteiros. Funcionando como uma *cache* de novas subscrições. Essas são mantidas até que seja possível obter *lugares vagos*. O número de colunas dessa matriz será também igual ao número de *agregados*. A informação guardada são os identificadores dos predicados de cada subscrição, porque os identificadores, como indicado, são "herdados" pelas subscrições previamente removidas.

Quando o número de subscrições removidas por cada *agregado* permite a adição das novas subscrições, um método obtém um vector de inteiros que indica as posições nos *agregados*. Esses dados são transferidos para o *device* onde está implementado um *kernel* que realiza a adição das novas subscrições. Mais uma vez, a sua implementação é apresentada na secção *CPU (Host)*.

As subscrições pendentes têm de ser tratadas no *host* até ao momento da sua transferência para o *device* para evitar a ocorrência de *falsos negativos*.

3.1.3. Eventos

Os eventos são conjuntos de pares da forma:

$$E_i = (Atributo, Valor \mid Lista\ Valores)$$

Os elementos contidos nestes pares são tratados de forma distinta e seguidamente será descrito o processo para a sua obtenção.

Os elementos atributos são obtidos através do mesmo processo que o indicado nos predicados, para manter a coerência do sistema no que diz respeito aos identificadores.

Usa-se um vector de *int2* para representar os eventos. Nele, está contida a informação de quantos pares (Atr, Valor) existem por cada evento. O valor *Y* e *Z* indicam a posição inicial e final, respectivamente, no *vector dos atributos*, funcionado como um *intervalo* de valores

representativos da lista de valores que o atributo tem. Assim garante-se que um atributo pode ter mais do que um valor. Nesta implementação apenas um evento é representado.

O *vector de atributos* indica o conjunto de atributos de todos os eventos, representados num *int3*. Um *int3* tem a seguinte informação:

Evento.x = Identificador do Atributo
Evento.y = Índice Inicial
Evento.z = Índice Final

Os *índices* têm um objectivo que advém dos eventos poderem ter mais do que um valor associado a cada atributo, um exemplo pode ser um texto com múltiplas *strings*. O texto é dividido em *k* valores, onde *k* é o número total de palavras que são tratadas de cada vez. Esta implementação não evita a redundância de valores entre os eventos, mas esse custo é amortizado com o pequeno número de eventos no sistema.

3.2. *GPU (Device)*

No *device*, todas as estruturas lineares apresentadas, e obtidas e transferidas do *host*, são mantidas na memória. A estrutura dos *agregados* ou os *valores dos predicados* são alguns dos exemplos. As únicas que não são mantidas são as estruturas dos eventos, porque são os únicos elementos que são voláteis em cada processo de filtragem. Volátil porque todo o processo de filtragem é realizado para eventos diferentes, reaproveitando os dados relativos às subscrições.

Aqui são mantidas em *memória global* duas estruturas que não são utilizadas no *host*, que correspondem aos *agregados* e à máscara de *bits*. A máscara é calculada aqui, enquanto os *agregados* são utilizados no processo de contagem, ou seja, no *kernel* correspondente.

A cada nova entrada de eventos a estrutura da *máscara* é actualizada. Tal é necessário porque a *máscara de bits* indica quais os predicados que foram aceites por um determinado evento. A estrutura da *máscara* é apenas um vector de inteiros com valores de 0, para rejeitado, a 1, para aceite. A avaliação da *máscara* é feita pelo *kernel máscara de bits*.

Nas secções seguintes vão ser apresentados os vários *kernels* desenvolvidos. Todos, partilham os mesmos pressupostos e portanto serão apresentadas as seguintes informações: os argumentos que o *kernel* recebe, qual ou quais as estruturas que são divididas para serem acedidas pelos *threads* e qual o seu funcionamento. Ao todo são usados quatro *kernels*.

Outro facto importante consiste na divisão do trabalho ser, em todos os *kernels*, igual. Foi sempre dividido de modo a que grupos de 32 *threads* contíguos efectuassem o mesmo trabalho, ou seja, o código processado por cada *thread* é sempre o mesmo. Assim, existe um aproveitamento do conjunto de *warps* formados em cada configuração do *tamanho da execução*.

Por consequência, o número de *threads* mínimo por *kernel* tem que ser sempre 32. Uma configuração com um número menor também não retira o real partido dos *GPUs*, porque os melhores resultados só são obtidos com um número relativamente grande de *threads*.

3.2.1. *Kernel Máscara de Bits*

Este *kernel* recebe como argumentos (ou parâmetros) as estruturas lineares dos predicados e dos eventos. O conjunto de estruturas para os *predicados* e eventos são:

Estruturas	Objectivo
Int3* Predicates	Nesta estrutura estão representados todos os predicados. Os valores X, Y e Z são os valores indicados na secção *predicados*.
Int* numberPredicates	Indica o número total de predicados a serem avaliados.
Float4* Values	Todos os valores distintos por entre os predicados.

Tabela 3.2 – Parâmetros representativos dos predicados

Estruturas	Objectivo
Int3* Events	Estão representados todos os atributos do evento. Os valores X, Y e Z são os valores apresentados na secção *eventos*.
Int2* sizeEvents	Estrutura indicativa de quantos atributos cada evento tem. O número de elementos é de apenas um, porque nesta implementação existe apenas 1 evento. O valor X indica a posição do primeiro atributo e a posição Y indica a do último.
Float4* values	Valores dos atributos.

Tabela 3.3 – Parâmetros representativos dos eventos

Como os *kernel* não devolvem qualquer tipo de valor, é necessário passar como parâmetro a estrutura onde se colocam os resultados do processamento. Logo, um parâmetro será um vector de inteiros. Essa será a estrutura da *máscara de bits*, residente na *memória global*.

A estrutura dividida para o trabalho é a dos predicados. Assim, a divisão de trabalho por *thread* será:

Número de predicados / Número de threads

O *kernel* percorre todos os predicados, um a um, e para cada verifica se os seus valores satisfazem o operador comparativamente com os valores do evento. Cada *thread* acede, inicialmente, ao predicado correspondente ao seu identificador global.

O índice do trabalho seguinte é obtido através da fórmula:

*TID_GLOBAL + Número de Warps * 32*

Este processo é efectuado até que o índice obtido pelo *thread* seja maior ou igual ao número total de predicados.

Na figura seguinte está exemplificado o processo efectuado a um predicado:

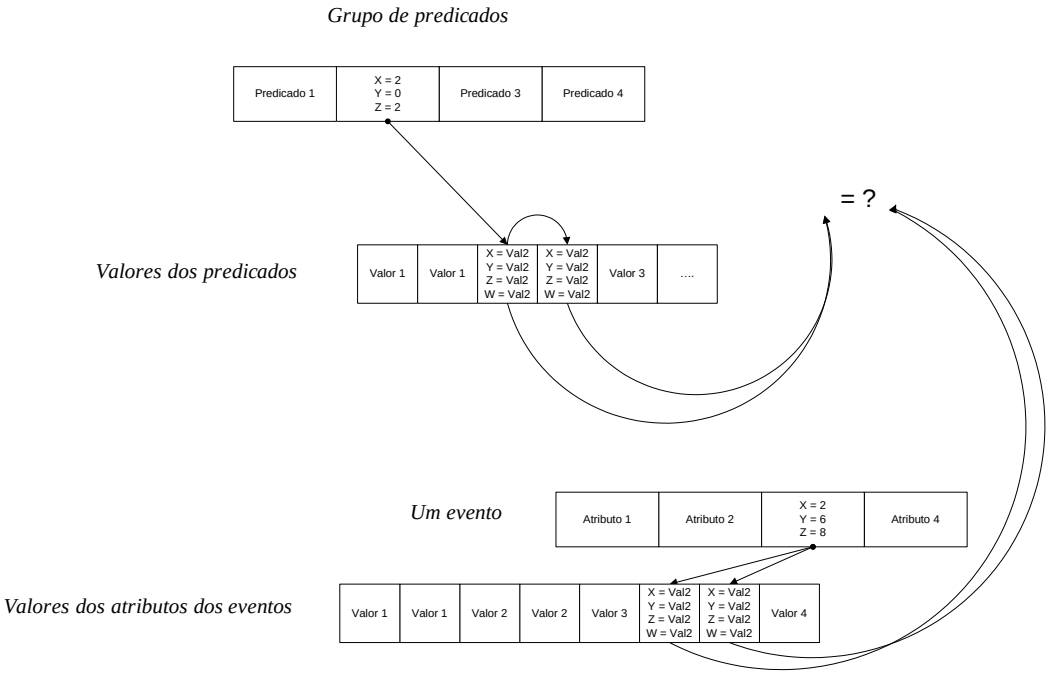


Figura 3.10 - Funcionamento do *kernel* para o operador (=)

No exemplo, o predicado com identificador dois tem como valor o segundo elemento no vector de *float4*. Como indicado na implementação das estruturas do predicado, cada elemento vale dois *float4*. O mesmo acontece no evento. No exemplo, tem quatro atributos e o *intervalo* de valores do terceiro é apenas um.

Na posição *Y* do predicado indica o operador 0, que identifica o operador de igualdade (=). No exemplo, existe a aceitação do predicado avaliado porque os valores satisfazem o operador, ou seja, são iguais tanto no evento como no predicado. Consequentemente, o valor da *máscara* na posição 2 é colocado a 1. Todo este processo é efectuado em todos os predicados.

Para cada predicado, os atributos de um evento são verificados enquanto não são encontrados valores que satisfaçam o operador. No *kernel*, um *switch* é utilizado para diferenciar os operadores e saber que operação efectuar.

A seguir é apresentado algum do código do *kernel*:

```
__global__ void
kernelBit1Event(int* tthreads, int3* predicates, int* n_predicates, float4* values, int3* anEvent, int2* sizeEvent, float4* valuesEvent, unsigned int* bitmap)
{
	const unsigned int tid = threadIdx.x;
	const unsigned int bid = blockIdx.x;
	const unsigned int pos = bid*blockDim.x + tid;
	const unsigned int nWarps = *tthreads/32;
	int oper;
	int atr;
	float4 val;
	float4 val2;

	for(int i = pos; i < *n_predicates; i+=(nWarps*32))
	{
		int3 currPred = predicates[i];
		val = values[currPred.z];
		val2 = values[currPred.z+1];
		oper = currPred.y;
		atr = currPred.x;
		int size = sizeEvent[0].y;
		for(int k = 0; k < size; k++)
		{
			int3 elemEvent = anEvent[k];

			if(elemEvent.x == atr)
			{
				for(int l = elemEvent.y; l < elemEvent.z; l+=2)
				{
					float4 currv = valuesEvent[l];
					float4 currv2 = valuesEvent[l+1];

					switch (oper)
					{
					case 0 :
						if(val.x == currv.x)
							bitmap[i] = 1;
						break;
						…
					}
				}
			}
		}
```

Listagem 3.1- *kernel* de obtenção da *máscara*

3.2.2. *Kernel* de Filtragem de Subscrições

No que se refere a esta parte da implementação, o *kernel* tem como objectivo a obtenção de todas as subscrições aceites por um dado evento. O número de invocações deste *kernel* é igual ao número de *agregados* porque o *kernel* foi desenvolvido para obter as subscrições aceites apenas para um *agregado* de cada vez. O *overhead* acumulado associado à invocação de vários *kernel* não é significativo, pois o número de *agregados* no sistema é sempre pequeno devido à utilização da exponenciação para definição do seu tamanho. Na implementação corrente o número total de *agregados* é de 7 e corresponde a subscrições com um máximo de 128 predicados.

Seguidamente, são apresentados todos os parâmetros recebidos:

Estrutura	Objectivo
Int* Subscriptions	Corresponde a uma coluna da matriz de *agregados*, ou seja, é um *agregado*. Nele, estão os identificadores de cada subscrição mais as referências para a *máscara de bits.*
Int* clusterSize	Indica o número total de elementos, ou seja: Número de Subscrições * (Potência de dois + 1 (Identificador de Subscrição))
Int* size	Uma potência de dois, para indicar qual o tamanho máximo por subscrição que o *agregado* recebe.

Tabela 3.4 – Parâmetros representativos dos *agregados*

Estrutura	Objectivo
Int* Bitmap	Corresponde à *máscara de bits* que foi obtida pelo *kernel* da *máscara de bits*. Esta estrutura está na *memória global*.

Tabela 3.5 – Parâmetro representativo da *máscara de bits*

Como foi anteriormente indicado, um *kernel* não devolve valores. O resultado é escrito numa estrutura passada como parâmetro. Neste *kernel* mais uma vez, a estrutura é um vector de inteiros. Esse vector tem o tamanho igual ao número total de subscrições no sistema. Cada posição é indexada pelo identificador único de cada subscrição. Os resultados possíveis são 0 ou 1, indicando se a subscrição não foi ou foi aceite respectivamente. A verificação de uma subscrição falha sempre que o valor de um dos predicado na *máscara de bits* é 0, evitando a desnecessária verificação dos seguintes.

Na figura seguinte é apresentado o funcionamento do *kernel*:

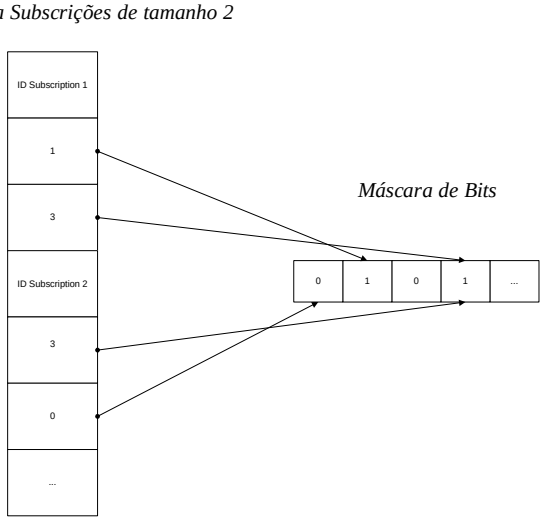


Figura 3.11 - Funcionamento do *kernel* para filtragem das subscrições

Na figura está representada uma invocação do *kernel* com o *agregado* das subscrições com tamanho 2. No exemplo, a primeira subscrição é aceite porque ambos os predicados referidos têm o valor 1. A subscrição seguinte falha porque um dos predicados é 0. Cada uma destas verificações é feita por um *thread*. A divisão de trabalho é feita através da divisão do número de subscrições do *agregado* pelos *threads.* Cada bloco de 32 *threads* analisa um conjunto de 32 subscrições contíguas.

A seguir é apresentado o código do *kernel*:

```
__global__ void
kernelMatch1Event(int* tthreads, unsigned int* d_bitmap, int* subs, int* clusterSize, int* size, unsigned int* result)
{
	const unsigned int tid = threadIdx.x;
	const unsigned int bid = blockIdx.x;
	const unsigned int pos = bid*blockDim.x + tid;
	const unsigned int s = (clusterSize[0]/(size[0]+1));
	const unsigned int nWarps = *tthreads/32;

	for(unsigned int i = pos; i < s;)
	{
		int match = 0;
		int ID = subs[i*(size[0]+1)];

		for(unsigned int j = 1; j <= size[0]; j++)
		{
			if(d_bitmap[subs[i*(size[0]+1)+j]]==1)
				match++;
			else
				break;
		}

		if(match == size[0]){
			result[ID] = 1;
		}

		i += (nWarps*32);
	}
}
```

Listagem 3.2- Código do *kernel* da contagem.

3.2.3. *Kernel* de Remoção de Subscrições

Para a remoção de um conjunto de subscrições é necessário saber em que posições e a que *agregados* cada uma das subscrições pertence. Esse processo é todo realizado no lado do *host* e já foi explicado na secção *Remover Subscrições.* Como foi indicado, é criado um vector de posições por cada *agregado* das subscrições a remover. Mais uma vez, para cada *agregado* diferente é invocado um *kernel*.

Os parâmetros que o *kernel* recebe são os seguintes:

Estrutura	Objectivo
Int2* subscriptionsToDel	Indica as *coordenadas* de cada subscrição a remover. O valor X indica o identificador do *agregado* (valor meramente informativo) e o valor Y indica a posição no *agregado* correspondente. O valor X do primeiro int2 indica quantas subscrições é necessário apagar.
Int* cluster	O *agregado* actualmente a ser alterado.
Int* n_predicates	Posição *m* na *máscara de bits* com valor 0. Corresponde ao *valor absorvente.*

Tabela 3.6 – Parâmetro representativos das subscrições a remover

Este *kernel* não tem como objectivo devolver um resultado mas sim alterar uma estrutura já existente em *memória global*. A estrutura alterada é a dos *agregados*. Aqui, não existe divisão de trabalho porque o número de subscrições para remoção não justifica, na maior parte das vezes, a divisão por múltiplos *threads*. Todo o trabalho é realizado apenas por um *thread*.

Da figura seguinte facilmente se retira o funcionamento do *kernel*:

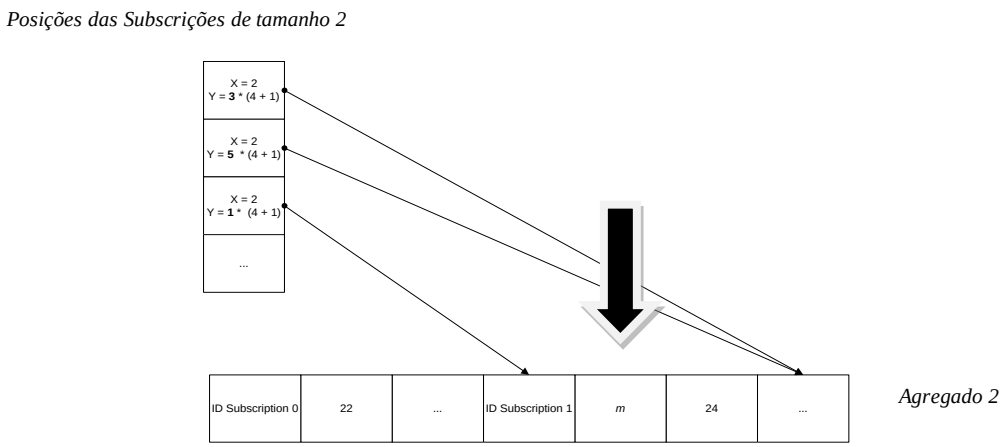


Figura 3.12 - Funcionamento do *kernel* para remoção de subscrições

Na figura, a seta indica a posição a alterar, da subscrição, para torná-la inválida. A posição corresponde ao primeiro predicado. Essa é alterada para o índice que aponta para o predicado "*false*" na máscara. No exemplo está representado o *agregado* 2 (para as subscrições com tamanho compreendido entre 3 e 4) e o vector das subscrições a apagar. Na posição *Y* desse vector está indicada a posição da subscrição a remover. O código do *kernel* é trivial:

```
__global__ void
kernelRemoveSubs(int2* subsToDel, int* cluster, int* n_predicates)
{
        for(int i = 1; i < subsToDel[0].x; i++)
                cluster[subsToDel[i].y+1] = n_predicates[0];
}
```

Listagem 3.3- Código do *kernel* da remoção de subscrições

3.2.4. *Kernel* de Adição de Subscrições

A adição de subscrições é um processo mais complicado que a remoção. Enquanto na remoção era necessário apenas a modificação de um valor, neste processo a modificação depende do número de subscrições e do seu tamanho. As operações neste contexto correspondem à escrita de cada nova subscrição na posição de subscrições anteriormente removidas.

Os parâmetros do *kernel* são:

Estrutura	Objectivo
Int2* subscriptionsToIns	Tem toda a informação das novas subscrições, como o identificador e predicados associados.
Int* cluster	O *agregado* actualmente a ser alterado.
Int* positions	Posições, no *agregado*, onde vão ser inseridas as subscrições. Cada posição corresponde a uma subscrição.
Int* numSubs	Variável indicativa do número total de subscrições a adicionar.
Int* size	Inteiro com valor igual a uma potência de dois. Indica qual o tamanho máximo de cada subscrição no *agregado*.

Tabela 3.6 – Parâmetro representativos das subscrições a adicionar

Mais uma vez, o objectivo do *kernel* não é devolver um resultado mas sim alterar os *agregados*, acrescentando as novas subscrições. Essa alteração passa basicamente por percorrer linearmente a estrutura das novas subscrições, juntamente com a estrutura das posições associadas a cada subscrição, e acrescentar no *agregado* as novas referências dos predicados.

Na figura seguinte está exemplificado todo esse processo:

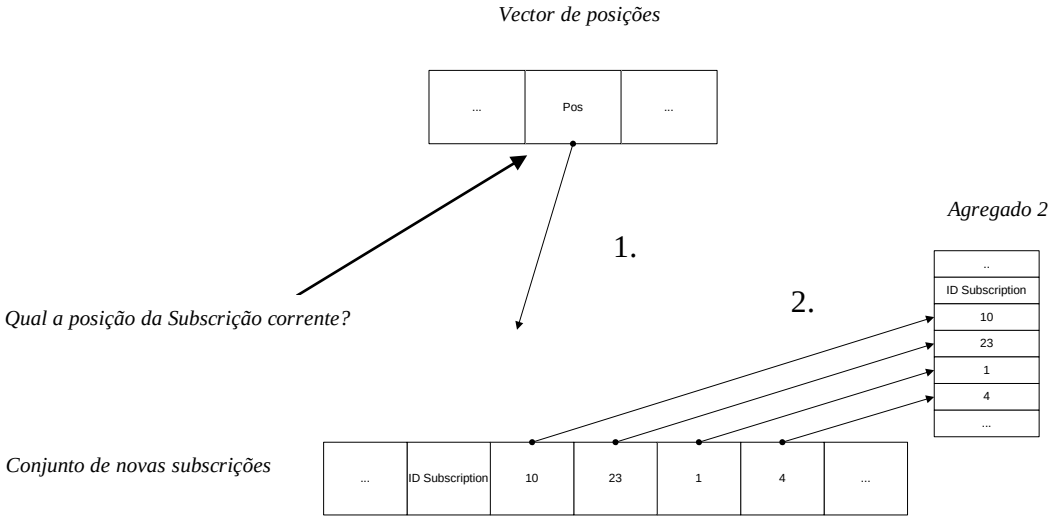


Figura 3.14 - Funcionamento do *kernel* para adição de subscrições

Os passos numerados são os seguintes:

1. Obtém-se a posição no *agregado* onde se vai escrever a nova subscrição.
2. Escrevem-se todos os predicados no *agregado*, introduzindo assim a nova subscrição.

Este processo é repetido para todas as subscrições.

O código do kernel é:

```
__global__ void
kernelInsertSubs(int* subsToIns, int* cluster, int* pos, int* numSubs, int* size)
{
        for(int i = 0; i < numSubs[0]; i++)
                for(int j = 1; j <= size[0]; j++)
                        cluster[pos[i]+j] = subsToIns[i*(size[0]+1)+j];
}
```

Listagem 3.4 - Código do *kernel* de adição de subscrições

3.3. Optimizações

A familiarização adquirida com a primeira implementação do algoritmo e a análise dos primeiros resultados permitiram e motivaram a exploração de diversas optimizações, descritas nas secções seguintes.

As principais centraram-se mais na exploração dos diferentes tipos de memórias existentes no *device*, entre as quais a *memória partilhada e memória constante*.

As outras incidiram mais na possibilidade de filtragem de mais eventos de uma vez. Concretamente, a solução foi modificada para permitir que 32 eventos sejam filtrados de uma só vez. Foi ainda incorporado, neste estudo, a filtragem de subscrições com *disjunções* de predicados.

Nas secções seguintes serão apresentadas todas as alterações fundamentais. Em primeiro lugar alterações que permitiram o uso da filtragem para 32 eventos, a seguir serão apresentadas as alterações a nível das memórias e, por último, a implementação do *kernel* e alterações nas estruturas do *host* para a possibilidade de filtragem das subscrições com *disjunções* de predicados.

3.3.1. 32 Eventos

Onde se localizava o maior desperdício, em termos de espaço em memória, era na obtenção dos eventos. Na implementação anterior, a filtragem era feita apenas para um evento e o *output* do *kernel* de filtragem era um vector de inteiros, que identificava cada uma das subscrições. Eram aceites as que tinham valor 1 e rejeitadas as que tinham valor 0. Mas, existe um desperdício de espaço subjacente nessa estrutura, porque eram apenas representados 0s e 1s tendo um funcionamento análogo a uma *máscara de bits*. Uma optimização possível foi a utilização de todos os bits desses inteiros. Assim, de uma só vez, obtêm-se todas as subscrições

aceites em 32 eventos. Essa optimização concretizou-se com recurso a operações *bitwise*. Foram definidas duas macros, uma para ler e outra para escrever nos bits de qualquer tipo de dados.

A seguir são apresentadas essas macros:

```
#define ACTIVE_BIT(i, j) (i & (1 << j))
#define EDIT_BIT(i, j) (i | (1 << j))
```

Listagem 3.5 - Código das *macros*

A primeira verifica apenas se bit *j*, no valor *i*, está a 1. A segunda macro modifica o bit *j*, no valor *i*, para 1.

Com estas macros, o acesso aos *bits* dos tipos de dados foi facilitado e a implementação desta nova optimização tornou-se acessível. Dois novos *kernels* foram necessários, um para a obtenção da nova *máscara de bits* e outro para a filtragem das subscrições.

No *host*, os eventos são representados nas mesmas estruturas apresentadas. O vector de *int3* tem os atributos de todos os eventos no sistema. O vector que indica o tamanho dos eventos tem agora 32 elementos. Os valores dos atributos são distribuídos pelo vector de *valores dos eventos* contiguamente.

3.3.1.1. *Kernel* da *Máscara de Bits*

A implementação é bastante semelhante ao kernel para 1 evento. Uma das diferenças reside num ciclo extra que percorre todos os 32 eventos. Esse ciclo é totalmente percorrido por *cada* predicado. Em relação aos parâmetros do *kernel* são os mesmos. Outra diferença reside na total utilização da estrutura *sizeEvents*, que contêm agora 32 eventos e não 1 como explicado na implementação inicial.

A seguir é apresentado o mesmo esquema da implementação anterior mas com a utilização da estrutura de eventos:

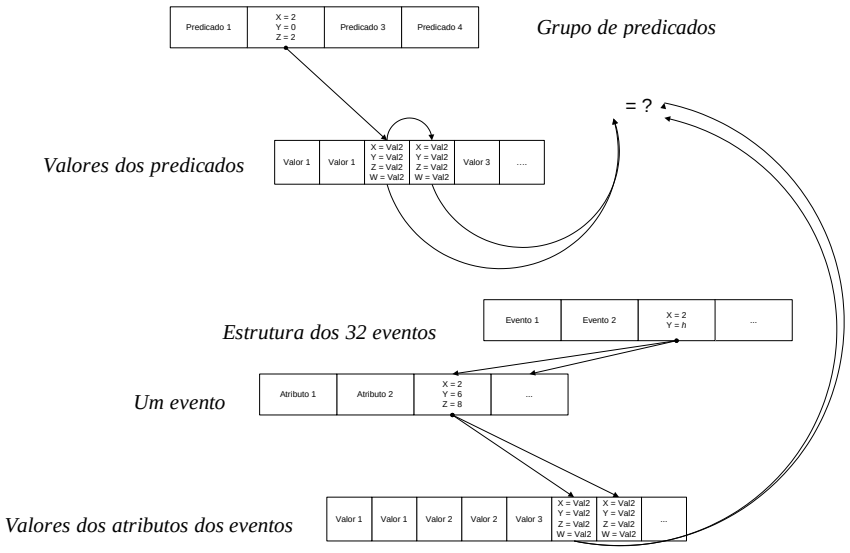


Figura 3.14 - Funcionamento do *kernel* de filtragem para subscrições para múltiplos eventos

Como indicado na figura, a estrutura dos eventos desta vez tem uma maior utilização. Por cada predicado processado, toda a estrutura é acedida. No exemplo, o evento 3 indica que os seus atributos começam na posição 2 do *vector de atributos* e acabam na posição *h*, perfazendo *h-2* atributos. Cada *bit* do inteiro, que representa um predicado, indica se um evento foi aceite. A modificação de cada *bit* é feita com recurso às macros referidas anteriormente.

Na figura seguinte está representada uma posição da *máscara de bits* (um inteiro), ou seja, um predicado *p*. Cada posição (*bit* = evento) indica se o predicado foi aceite para o evento representado nessa posição:

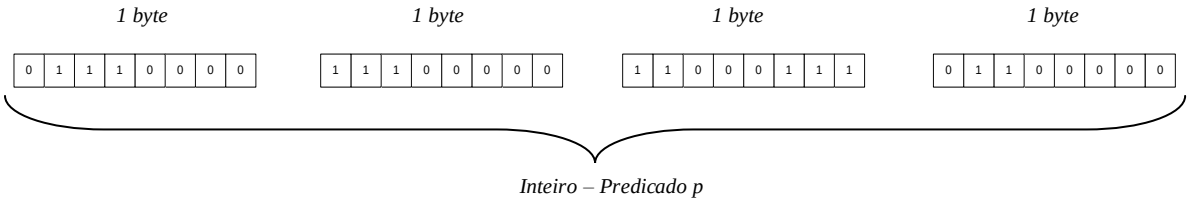


Figura 3.15 - Representação de um inteiro de 32-bits (predicado)

Na figura estão representados os 4 *bytes* de um predicado. Os *bits* com valor 1 são os eventos que são satisfeitos pelo predicado analisado. Toda a *máscara* gerada é mantida na *memória global*, como se verificava na implementação anterior.

O código acrescentado ao novo kernel é:

```
…
for(unsigned int j = 0; j < 32; j++)
{
        int2 Event = sizeEvent[j];

        for(int k = Event.x; k < Event.y; k++)
        {
                por cada caso do switch
                bitmap[i]=EDIT_BIT(bitmap[i],j);
                …
        }
…
```

Listagem 3.6 - Código do novo *kernel* do cálculo da *máscara*

3.3.1.2. *Kernel* de Filtragem de Subscrições

Como agora a informação que está contida na *máscara de bits* está representada de uma forma completamente diferente, o *kernel* para a filtragem também tem de ser compatível com a leitura dessa nova informação. A alteração simplificou o *kernel* e tornou-o mais eficiente. O novo código requer menos expressões condicionais reduzindo a probabilidade do fluxo de execução de cada *thread* divergir, o que é muito penalizado na arquitectura do *GPU*.

O *output* do *kernel* também é diferente. Também é um vector de inteiros, também cada posição representa uma subscrição, mas agora cada *bit* do inteiro está associado ao evento

correspondente. Esse *output* é obtido através do *and* lógico em cadeia dos *bits* de cada predicado da subscrição. Os parâmetros são iguais porque a estrutura dos *agregados* não foi modificada. A seguir é apresentado o código do novo *kernel*.

```
__global__ void
kernelMatch32Events(int* tthreads, unsigned int* d_bitmap, int* subs, int* c, int* size, unsigned int* result)
{
                const unsigned int tid = threadIdx.x;
                const unsigned int bid = blockIdx.x;
                const unsigned int pos = bid*blockDim.x + tid;
                const unsigned int s = (c[0]/(size[0]+1));
                const unsigned int nWarps = *tthreads/32;

                for(unsigned int i = pos; i < s;)
                {
                        int ID = subs[i*(size[0]+1)];
                        unsigned int match = d_bitmap[subs[i*(size[0]+1)+1]];

                        for(unsigned int j = 2; j <= size[0] && match != 0; j++)
                                match &= d_bitmap[subs[i*(size[0]+1)+j]];

                        result[ID] = match;

                        i += (nWarps*32);
                }
}
```

Listagem 3.7 - Código do novo *kernel* do cálculo da *filtragem*

3.3.2. Memórias

Uma das possibilidades de melhoramento dos resultados obtidos na implementação anterior passou pela utilização de outras memórias que têm como benefício o facto do custo dos acessos ser mais barato do que se verifica na *memória global*. As memórias citadas são a *partilhada* e a *constante*. Porém, existem algumas limitações em ambas as memórias sobretudo na sua dimensão.

Nas próximas subsecções ambas as memórias são apresentadas tais como as alterações efectuadas a nível de implementação para a possibilidade do seu uso.

3.3.2.1. *Memória Partilhada*

O tamanho máximo de dados que podem ser guardados nesta memória é de 16 *KBytes*. Além da reduzida dimensão, esta memória apenas pode ser acedida por um *bloco* de *threads* de cada vez. Em contrapartida, tem um tempo de acesso consideravelmente mais reduzido pelo que existe interesse em poder ser usada para guardar dados que são repetidamente acedidos. Uma estrutura que é repetidamente acedida é a *máscara de bits*, no processo de filtragem. Portanto, um ganho pode ser retirado se essa estrutura permanecer em *memória partilhada*. Como se espera que o número de predicados seja um número elevado, na ordem das dezenas de milhar, a sua compactação é necessária. Ela foi conseguida através da representação dos predicados em *bits* individuais. Assim, usando um inteiro pode-se representar 32 predicados de uma só vez.

Para um exemplo de 10000 predicados seriam necessários apenas:

Número de Inteiros = 10000 / 32 = 313, dos 4096 disponíveis.

Com a redução da ocupação da *máscara* em aproximadamente 97%, existe a possibilidade real de utilização desta estrutura em *memória partilhada*.

O cálculo da nova *máscara* é análogo ao anterior. Quando se altera um valor, em vez de se modificar um inteiro modifica-se um *bit*. Isso foi conseguido através do uso de operações *bitwise*.

Na filtragem, o processo também era análogo, alterando apenas o método de acesso à estrutura, que agora é acedida a nível do *bit*.

Foram utilizadas as mesmas macros que foram desenvolvidas na implementação para 32 eventos, a leitura é feita com a macro *ACTIVE_*BIT e a escrita pela macro *EDIT_BIT*. Com esta alteração, a escrita da *máscara* na *memória partilhada* é realizada apenas *b* vezes. O valor *b* é igual ao número total de blocos lançados no *tamanho da execução*.

No *kernel*, inicialmente, o *thread* com identificador 0 de cada *bloco* escreve para *memória partilhada* a *máscara*. Todos os *threads*, pertencentes ao seu *bloco*, acedem assim a mesma estrutura com um custo bastante mais reduzido. A inicialização e leitura da estrutura são feitas da forma:

```
extern __shared__ unsigned int b[];

if(tid == 0)
{
        for(int i = 0; i < bits; i++)
                b[i] = d_bitmap[i];
}
__syncthreads();
```

Listagem 3.8 - Código para colação de dados na *memória partilhada*

O código apresentado é acrescentado no *kernel* de filtragem, sendo essa apenas a única alteração a nível estrutural do *kernel*. O acesso à estrutura na *memória partilhada* é realizado com recurso à macro apresentada anteriormente). O método *__syncthreads()* tem como objectivo ser uma barreira de sincronização entre todos os *threads* do mesmo *bloco*. Com esta barreira, é garantido que os *threads* só acedem à estrutura na *memória partilhada* quando esta estiver totalmente escrita.

Na implementação dos 32 eventos, esta optimização não é possível. Porque os inteiros de cada predicado têm um papel fundamental, já que representam os 32 eventos. Mas, na obtenção da *máscara*, esta optimização é implementada. A estrutura com os valores dos eventos é escrita para a *memória partilhada* porque é a estrutura mais acedida pelos *threads* no processo de obtenção da *máscara de bits*.

O código da alteração é apenas:

```
if(tid == 0)
{
        for(int i = 0; i < 32; i++)
        {
                int2 Event = sizeEvent[i];

                for(int k = Event.x; k < Event.y; k++)
                {
                        int3 elemEvent = anEvent[k];

                        for(int l = elemEvent.y; l < elemEvent.z; l+=2)
                        {
                                valuesEvent[l]= vEvent[l];
                                valuesEvent[l+1]= vEvent[l+1];
                        }
                }
        }
}
__syncthreads();
```

Listagem 3.9- Código para colação de dados na *memória partilhada* (2)

O primeiro *thread* de cada *bloco*, mais uma vez, efectua a leitura de todos os valores dos eventos para a *memória partilhada*, funcionando quase como *cache* da *memória global*. O método de sincronização tem o mesmo objectivo.

3.3.2.2. *Memória Constante*

Esta memória tem a característica fundamental de ser *cached*. Assim é garantido que todos os acessos anteriormente feitos são mantidos nos 8 Kbytes disponibilizados como *cache*. Nesta memória podem ser guardados até 64 KBytes de dados.

O *device* não tem permissão para escrever. Se se pretender adicionar dados nela apenas o *host* o pode fazer. Como tal, as únicas estruturas que podem tirar real partido desta memória são as utilizadas para a obtenção da *máscara*, como os vectores de *valores dos eventos* ou o vector do *tamanho dos eventos*. Outra razão prende-se com o facto de essas estruturas serem totalmente acedidas por *cada* predicado existente no sistema, com o objectivo de se saber se foi aceite ou não.

A implementação efectuada residiu na introdução dos eventos, dos atributos dos eventos e dos valores dos eventos em *memória constante*. Todas as alterações foram realizadas no *host*. Em todas as variáveis foram adicionados os qualificadores *__device__ __constant__*. Para comunicar essas estruturas para o *device* foi utilizado o método *cudaMemcpyToSymbol*.

3.4. Disjunções e Negações de Subscrições

O algoritmo original [17] só suporta subscrições que são conjunções de predicados. Um dos melhoramentos implementados consistiu em suportar disjunções de predicados e disjunções de conjunções de predicados.

Em primeiro lugar, o tratamento das subscrições com disjunções tem de ser diferente. Esse tratamento passou pela decomposição das subscrições, ou seja, uma subscrição com a seguinte constituição:

$$S_1 = P_1 \vee P_2 \wedge P_3 \vee P_4$$

Será convertida em:

$$S_i = P_1$$
$$S_{i+1} = P_2 \wedge P_3$$
$$S_{i+2} = P_4$$

Basicamente, uma subscrição com *n* disjunções será sempre dividida em *n+1* subscrições. Na forma conjuntiva dos predicados, o tratamento das subscrições adquiridas na decomposição é realizado como se subscrições individuais se tratassem. No exemplo, para a *S1* ser aceite *pelo menos* uma das subscrições, da decomposição, tem de ser aceite.

A solução passou pela criação de um novo *agregado especial* no *host*. Nesse, vão estar os identificadores das subscrições criadas na decomposição. No exemplo seria inserido numa das colunas do *agregado especial* os identificadores Si, Si+1 e Si+2. O *agregado especial* seria algo como:

S1	...
Si	..
Si+1	...
Si+2	...

Figura 3.16 - Representação do *agregado especial*

Depois de preenchida, a estrutura é transferida para o *device* para ser processada e para serem obtidas todas as subscrições aceites, através de um novo *kernel*. Este novo *kernel* é invocado depois do *kernel* da filtragem. Essa ordem de invocação é fundamental porque o resultado da filtragem destas novas subscrições depende directamente dos resultados da filtragem das subscrições, obtidas na decomposição. Essa dependência verifica-se porque, como já foi indicado, uma subscrição com disjunções é aceite quando *uma qualquer* subscrição, obtida na decomposição, for satisfeita.

A seguir, é apresentado o *kernel* implementado para essa possibilidade, na implementação para 32 eventos:

```
__global__ void
kernelMatch32EventsDisjunctions(int* tthreads, unsigned int* subsAccepted,unsigned int* subs, int* numSubs, int* size, unsigned int* result)
{
                const unsigned int tid = threadIdx.x;
                const unsigned int bid = blockIdx.x;
                const unsigned int pos = bid*blockDim.x + tid;
                const unsigned int nWarps = *tthreads/32;

                for(unsigned int i = pos; i < *numSubs;)
                {
                        int ID = subs[i*(size[0]+1)];
                        unsigned int match = subsAccepted[subs[i*(size[0]+1)+1]];

                        for(unsigned int j = 2; j <= size[0]; j++)
                                match |= subsAccepted[subs[i*(size[0]+1)+j]];

                        result[ID] = match;

                        i += (nWarps*32);
                }
}
```

Listagem 3.10 - Código para filtragem de subscrições do *agregado especial*

Outra funcionalidade implementada foi a possibilidade de filtragem de negações de conjunções. A única diferença residiu na utilização de um número negativo no lugar do identificador da subscrição. Por exemplo, uma subscrição com o identificador associado 10 passaria a ter o identificador com o valor -10, assinalando que a subscrição é uma negação. Quando, no *host*, se obtêm os resultados relativos a cada subscrição, verifica-se o seu identificador e, caso seja negativo, o resultado é negado.

4. Validação Experimental

A filtragem de eventos em sistemas editor/assinante é um ponto crítico no desempenho do sistema. No algoritmo escolhido, esse processo está dividido em duas fases distintas. A obtenção da *máscara de bits*, que indica quais os predicados satisfeitos por um determinado evento, e a subsequente fase de contagem para determinar as subscrições que foram aceites.

Na implementação descrita anteriormente, essas duas fases são obtidas pela execução sequencial de dois *kernel* diferentes. Nesta secção é efectuado um estudo para determinar as principais características que afectam o seu funcionamento e desempenho. Para o correcto entendimento dos resultados, foram efectuados vários testes nas implementações desenvolvidas. Nessa análise experimental, de ambas as fases, foram utilizadas várias configurações de *threads* para a distribuição do trabalho. Foram ainda testadas as variantes do algoritmo no que diz respeito ao tipo de memórias utilizadas.

É importante salientar os pontos críticos na execução dos *kernel*, pois existem vários condicionantes que afectam o seu funcionamento e desempenho. Um *kernel* pode não ser invocado se o número de *threads* pedidos para serem utilizados necessitar de mais registos do que os disponíveis por Multiprocessador (*SM*). Outra impossibilidade pode ser a requisição de mais *memória partilhada* do que a disponível. Como estes dois condicionantes são partilhados pelos *blocos*, a execução destes é também afectada e impossibilita a execução.

O fabricante proprietário desta *API* oferece uma fórmula que permite, de antemão, obter qual o número de registos que são necessários por *bloco:*

$$^{1}ceiling\left(ceiling(\text{número de Warps}, 2) * NumRegPerThread * 32, \frac{8192}{32}\right)$$

Desta fórmula, retira-se que o número permitido de *blocos activos* é fortemente determinado pela complexidade do *kernel*, ou seja pelo número de registos que cada *thread* necessita e, ainda, pelo número de *warps* de cada *bloco*, que são sempre lançados aos pares e que necessitam no mínimo de 256 registos.

Os testes foram realizados em dois *GPUs* diferentes com o objectivo de serem apresentados resultados mais fedignos e que fossem capazes de sustentar as conclusões retiradas. Também foram realizados no *CPU*, de forma a possibilitar a comparação directa da performance do *GPU* face ao desempenho de um *CPU* da mesma categoria.

[1] Arredonda para cima, para um valor múltiplo do segundo argumento

4.1. Configuração dos testes

As características dos *GPUs* e do *CPU*, de uma forma sucinta, são:

	Características *device 1*	Características *device 2*	Características *CPU*
Modelo	8400M G (1 *SM*)	8800 GT (14 *SM*)	Core 2 Duo – T5250
Memória RAM	256 MBytes	512 MBytes	2 GBytes
Velocidade Clock (por *SM*)	0.80 GHz	1.6 GHz	1.5 Ghz Velocidade FSB: 667 Mhz Cache L1: 32 Kbytes/ 32 Kbytes Cache L2: 2 Mbytes
Sistema Operativo	Windows Vista 32-bit SP1	Windows Vista 32-bit SP1	Windows Vista 32-bit SP1

Tabela 4.1- Especificações das unidades de processamento e sistema operativo utilizado nas máquinas que as continha

A distribuição das estruturas e tamanho das configurações nos testes foram:

Número de blocos: 1…10 (até 20 no cálculo da *máscara*)

Número de threads por bloco: 32…512 ($\mathbf{2^5} \ldots \mathbf{2^9}$)

Número de *agregados:* 7

Número total de Subscrições: 500000

Número de iterações por configuração: 100

O número total de subscrições é 500 mil e são distribuídas pelos *agregados* de acordo com a seguinte fórmula:

$$^{2}floor\left(500000 * 2^{-i}, 32\right), de\ i \in [1, 2, 3, \ldots, 6]$$

São distribuídas as restantes pelo *agregado 7*. Assim, a distribuição concreta das subscrições está representada na seguinte tabela:

Número de *Agregado*	Número de Subscrições
1	249984
2	124992
3	62496
4	31232
5	15616
6	7808
7	7872

Tabela 4.2- Distribuição das subscrições pelos *agregados.*

[2] Arredonda por defeito para um valor múltiplo do segundo argumento

Na falta de um *benchmark* normalizado, assume-se que em sistemas deste tipo as subscrições em maior número têm poucos predicados, havendo uma diminuição progressiva do número de subscrições com o aumento do seu *tamanho*.

Pela mesma razão, as configurações avaliadas, no que diz respeito à composição das subscrições, a predicados aceites por cada evento, número de atributos por evento, etc, são obtidas de forma aleatória, usando sempre uma distribuição uniforme, obtida à custa da função *rand()* do *ANSI-C*.

4.2. Introdução e Remoção de Subscrições

Para permitir que o sistema seja dinâmico, é necessário garantir a entrada e saída de novas subscrições. De seguida, são apresentados os resultados obtidos pelos *kernel* de adição e remoção de subscrições. Esses só correm com um *thread* e é de realçar que estes testes foram apenas realizados no *device 1*.

O estudo realizado consistiu no envio do mesmo número de subscrições para cada *agregado*, ao mesmo tempo, e os resultados obtidos foram:

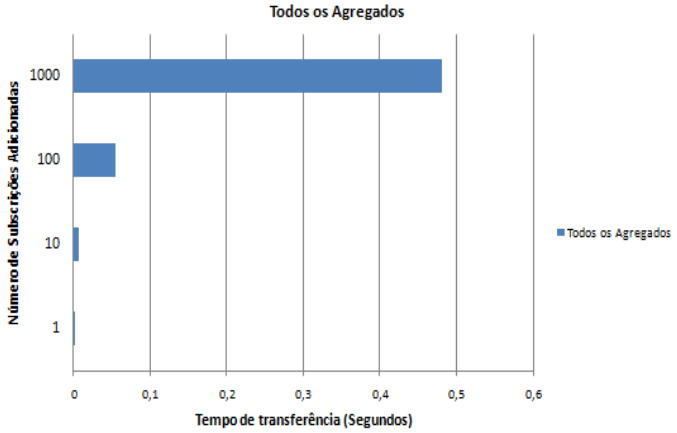


Figura 4.1- Resultados da adição de subscrições por todos os *agregados*. De salientar que os valores correspondem à inserção de subscrições em todos os *agregados*, ou seja, os valores correspondem a 7 vezes mais subscrições que as indicadas.

Os resultados obtidos indicam claramente, como seria de esperar, que quantas mais subscrições e quanto maior for o tamanho máximo do *agregado* maior é o tempo de execução.

Os resultados obtidos mostram que o tempo, como esperado, cresce com a dimensão dos dados a transferir, mas não é estritamente proporcional para quantidades pequenas. Isto pode ser explicado pelo *overhead* do processo de comunicação, uma vez que nesta experiência não há impacto da paralelização, já que foi utilizada apenas um *thread*.

Operação	Tempo de execução (Segundos)
Tempo de Inserir 1 Subscrição	0,0029
Tempo de Inserir 10 Subscrições	0,0082 (2.8x)
Tempo de Inserir 100 Subscrições	0,0551 (18,7x)
Tempo de Inserir 1000 Subscrições	0,4813 (164x)

Tabela 4.3- Resultados obtidos na adição.

Para o processo de remoção, foi apenas testada a remoção de 1, 10, 1000 e 10000 (para os possíveis) em *cada agregado.* O gráfico obtido foi:

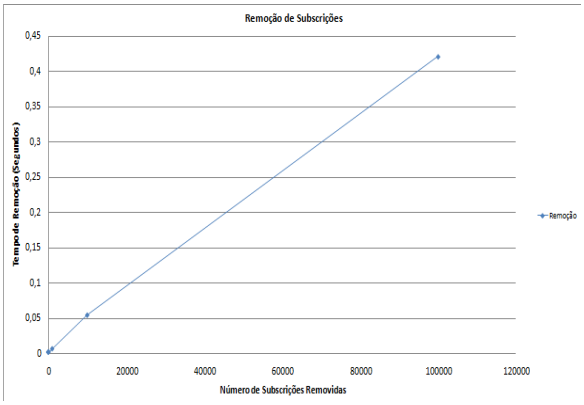


Figura 4.2- Gráfico dos resultados obtidos na remoção de subscrições.

Nota-se uma evolução claramente linear do custo da remoção em função do trabalho realizado.

Os resultados obtidos, vistos em conjunto, mostram que o custo de transferências das subscrições e a sua remoção não é desprezável. Ou seja, não é viável transferir o estado todo do *host* para o *device.* Isto reforça a ideia que a actualização do *device* de forma incremental é uma necessidade.

4.3. 1 Evento

Existem duas fases, a obtenção da *máscara* e a contagem dos predicados aceites das subscrições.

Em ambas, numa fase inicial, o estudo dos resultados incide na comprovação da fórmula fornecida pelo fabricante. Esse estudo é realizado em ambas as memórias, tanto na *global* como na *constante* para também demonstrar se efectivamente há um ganho no uso da *memória constante*. No processo de contagem, o estudo foi efectuado na memória *global* e *partilhada*.

Para todos os testes o número de predicados foi: 1000, 10000 e 30000. Esta variação vai tentar provar qual será o impacto do número de predicados no sistema no processo de filtragem.

De seguida, é apresentada uma tabela com toda a informação relevante dos *kernel* utilizados:

Kernel	Número de Registos por *thread*	Tamanho memória partilhada
Cálculo Máscara de bits (Global)	20	-
Cálculo Máscara de bits (Constante)	20	-
Contagem (Global)	8	-
Contagem (Partilhada)	10	sizeof(int)*ceil((número de predicados+2)/32)

Tabela 4.4- Características dos *kernel.*

4.3.1. Cálculo da *Máscara de Bits*

O processo de geração de eventos e predicados foi definido à partida. Ambos os elementos são gerados aleatoriamente utilizando a função *rand*() do *ANSI-C*. As estruturas serão as mesmas indicadas na secção *implementação* e cada elemento predicado terá o atributo igual ao seu identificador, o operador igual a um número aleatório até 7 e o elemento valor terá um valor aleatório entre um e o número de predicados existentes. Os eventos terão como atributos um valor aleatório com valor máximo igual ao número de predicados e o valor será mais uma vez um valor aleatório.

4.3.1.1. *Memória global*

O *kernel* deste teste necessita de 20 registos. Através da fórmula indicada pelo fabricante, deduz-se que, para cada *tamanho* dos *blocos,* obtém-se, como melhor configuração os seguintes dados:

Número de *threads* por *bloco*	Número de registos por *bloco*	Número máximo de *threads* activos	Melhor número de *blocos*
32	1280	192	6
64	1280	384	6
128	2560	384	3
256	5120	256	1

Tabela 4.5- Melhores configurações para o *kernel*

Por exemplo, os *blocos* com 32 *threads* terão como registos:

$$\boldsymbol{ceiling}\left(\boldsymbol{ceiling}(\mathbf{1},\mathbf{2}) * \mathbf{20} * \mathbf{32}, \frac{\mathbf{8192}}{\mathbf{32}}\right) = ceiling(\mathbf{2} * 20 * 32, \mathbf{256}) = ceiling(\mathbf{1280}, 256)$$

Obtendo-se assim o valor 1280, que é múltiplo de 256. O mesmo processo é realizado para todos os *tamanhos.*

Para a obtenção do número máximo de *threads activos,* o processo foi directo. Com o valor do número de registos necessários por *bloco,* pode-se deduzir o número máximo de *blocos* através da fórmula:

$$floor(\frac{8192}{NumRegPerBlock})$$

Retira-se da fórmula que o número máximo de *blocos* é $floor\left(\frac{8192}{1280}\right) = 6$. O número máximo de *threads activos* será então 6 * 32 = 192. Este processo é considerado sempre igual daqui para a frente.

Os resultados obtidos na experiência com 30000 predicados permitiram obter o seguinte gráfico:

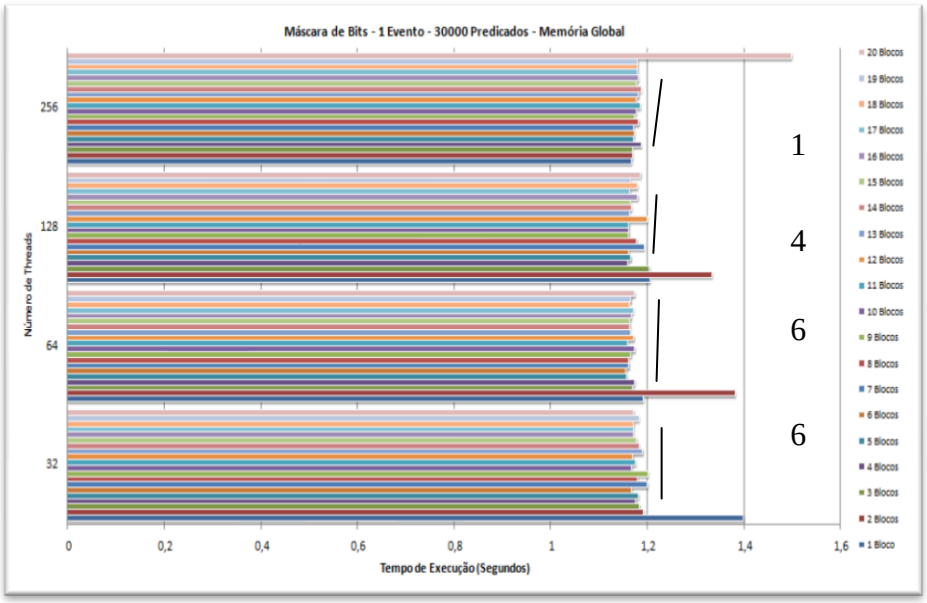


Figura 4.3- Gráfico dos resultados obtidos na obtenção da máscara para 30000 predicados.

De uma forma geral, os resultados obtidos são bastantes semelhantes. Numa análise mais fina, constata-se que, como indicado na figura, o número máximo de *blocos* máximo "permitido" pelos registos obtiveram os melhores tempos. Os melhores valores foram exactamente os deduzidos, tirando o resultado para o *tamanho* 128, mas a variação foi mínima, sendo apenas 4% inferior ao melhor valor. O melhor tempo obtido foi **1,15539** segundos para a configuração (6, 64).

No gráfico também estão representadas rectas que demonstram o declive da relação entre os resultados. Esse comportamento pode ser explicado através do número máximo de *blocos* que podem correr *activamente*, ou seja, o declive da recta torna-se maior à medida que o número de *blocos* limitados (pelos registos) for menor e quantos mais forem lançados mais é verificado o *overhead* de comutação de *threads*.

Os picos verificados foram casos isolados, porque foram realizadas outras experiências que demonstraram que esse comportamento foi um caso isolado.

No exemplo para 10000 predicados tal constatação é reforçada:

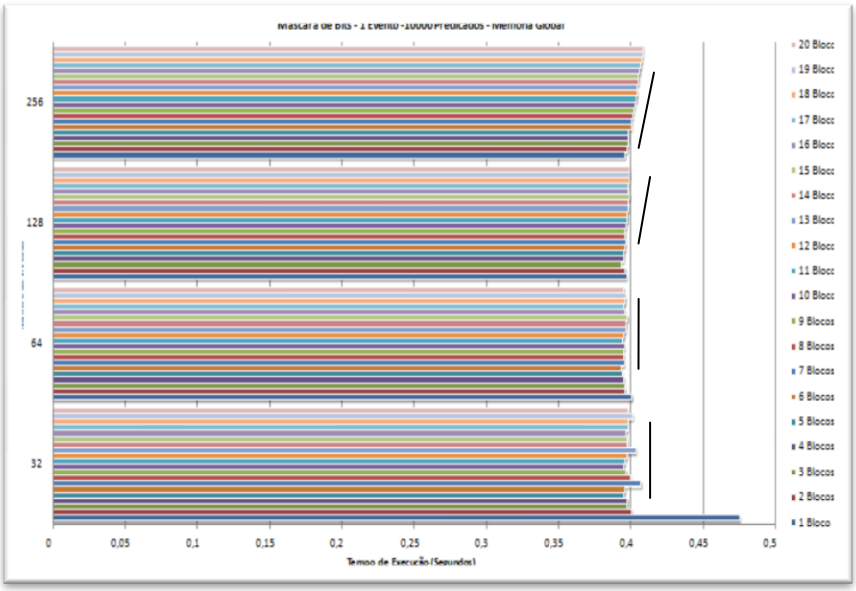


Figura 4.4- Gráfico dos resultados obtidos na obtenção da máscara para 10000 predicados.

Outra ilação que se pode retirar destes resultados é que os tempos de execução dependem directamente do número de predicados utilizados. Como se verifica, os melhores tempos obtidos variam linearmente com o número de predicados. No exemplo de 30000 predicados o melhor tempo foi **1,15539** segundos, espera-se que o tempo para 10000 seja 1/3 menor, ou seja, por volta de (arredondando para 1,12) 0,4 segundos. Efectivamente o melhor tempo para 10000 predicados foi **0,393433** segundos, mostrando assim que a variação é linear. O mesmo verificou-se para 1000 predicados, onde o melhor tempo se localizou nos **0,0489023** segundos, ou seja, aproximadamente 1/10 do tempo de 10000 predicados. De notar que todos os tempos correspondem a 100 invocações do mesmo *kernel*, ou seja, uma invocação do *kernel* corresponde a 1/100 dos tempos obtidos.

Assim, pode-se concluir o número de registos por *thread* pode afectar o débito do *kernel*. Porque, quanto maior for esse número mais difícil se torna o uso de múltiplos *threads activos*. Os resultados em si não variam, mantendo-se praticamente constantes, porque por mais *threads* que sejam pedidos para serem invocados, no máximo só podem correr os indicados na tabela. Quantas mais forem invocadas, mais *overhead* de escalonamento é acrescido, validando as conclusões retiradas da fórmula fornecida pelo fabricante. A semelhança dos resultados para as diversas configurações apoia a conclusão que a limitação ao desempenho estará nos acessos à memória.

Em relação ao *device 2*, os resultados para 30000 predicados foram:

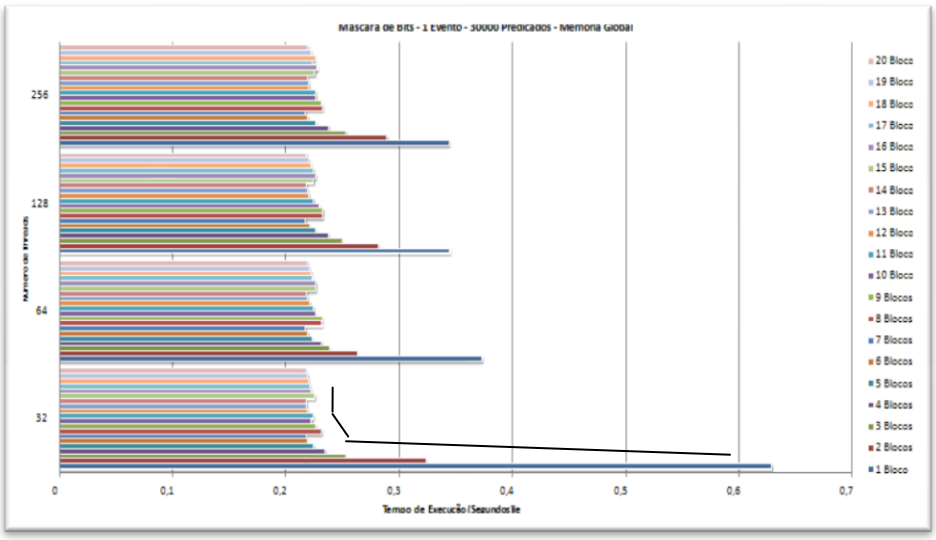


Figura 4.5- Gráfico dos resultados obtidos na obtenção da máscara para 30000 predicados, no *device 2*.

O comportamento indicado pelas rectas é explicado através do número de *SM* que o *device 2* oferece. Como indicado, o *device* tem 14 *SM*. Na figura, nota-se uma clara tendência para o melhoramento dos resultados quando o número de blocos é um múltiplo de 14. Os melhores resultados são, em todos os casos, obtido ou com 7 ou com 14 blocos. Isso é explicado porque com uma distribuição uniforme pelos *SM* disponibilizados, os resultados obtidos são melhores.

Verifica-se também um *speed-up*, obtidos com a introdução de novos MP no processamento, havendo a partir dos 7 blocos uma clara tendência para que os resultados obtidos sejam semelhantes. Esse comportamento pode ser explicado através da granularidade da tarefa que, à medida que mais *threads* são introduzidos, menor ela se torna. O melhor tempo obtido neste teste foi **0,217192** segundos. Os melhores tempos foram **0,0772576** segundos para 10000 e **0,0146118** segundos para 1000 predicados. Para ambos os gráficos, os comportamentos verificados foram os mesmos, apenas no caso de 1000 predicados as oscilações foram maiores mas mais uma vez explicado pelo facto dos tempos de execução serem tão pequenos que qualquer oscilação é sentida. Estes resultados mostram que o número de multiprocessadores ajuda na redução do tempo de cálculo, mas acabam por se manifestar as mesmas limitações dos acessos à memória.

4.3.1.2. *Memória Constante*

Os registos, por *thread*, do *kernel* para a obtenção da *máscara* são os mesmos 20. Portanto, espera-se que as mesmas configurações sejam as que têm melhores resultados.

Para o *device 1*, com 30000 predicados, o gráfico obtido foi:

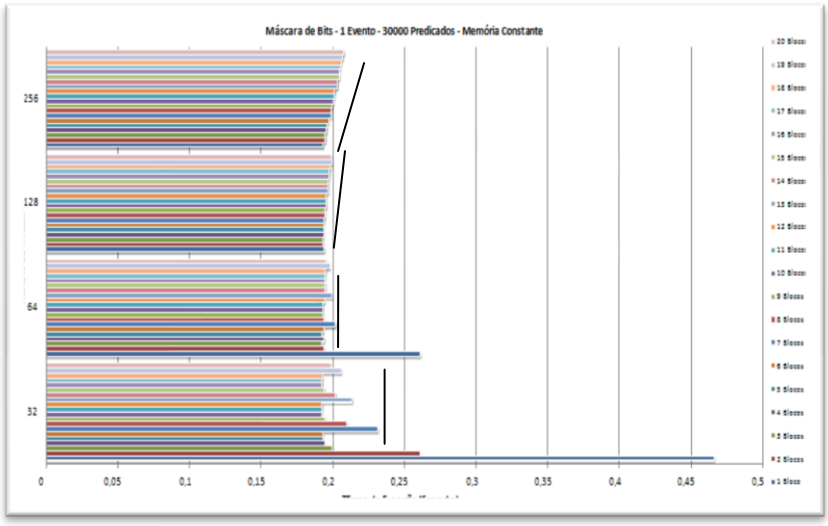


Figura 4.6- Gráfico dos resultados obtidos na obtenção da máscara para 30000 predicados, com memória constante.

Relembrando, o melhor tempo com *memória global* para 30000 predicados foi **1,15539** segundos e agora o melhor tempo obtido foi **0,192445** segundos! Houve um ganho efectivo da ordem das **6 vezes**. Em relação ao comportamento dos resultados, mais uma vez, os valores estão bastante próximos uns dos outros. O declive das rectas tem o mesmo comportamento devido ao número de registos, confirmando as conclusões da secção anterior. Para 1000 e 10000 predicados os gráficos obtidos têm o mesmo comportamento. Em relação aos melhores tempos, para os 10000 predicados foi **0,0715265** segundos (aumento de **5.5x**) e para 1000 foi **0,0163041** segundos (**3x**).

Em relação ao *device 2*, o mesmo comportamento foi verificado. Os melhores tempos foram obtidos com o número de *blocos* múltiplo de 14. As maiores oscilações são mais uma vez visíveis no teste de 1000 predicados, com a mesma justificação. Os melhores tempos obtidos foram: para 30000 predicados **0,0238108** segundos (aproximadamente 9x de ganho em relação à mesma implementação mas em *memória global*), para 10000 foi **0,0137242** segundos (**5.6x**) e por último para 1000 foi de **0,00796163** segundos (**1.8x**).

Como conclusão, apresentam-se duas pequenas tabelas com os melhores resultados e indicações de quais foram os ganhos, em ambos os *devices*:

30000 Predicados		10000 Predicados		1000 Predicados	
Global (segundos)	Constante (segundos)	Global (segundos)	Constante (segundos)	Global (segundos)	Constante (segundos)
1,155	0,1924	0,3934	0,0715	0,0489	0,0163
100%	6x	100%	5.5x	100%	3x

Tabela 4.6- Melhores resultados no *device 1*

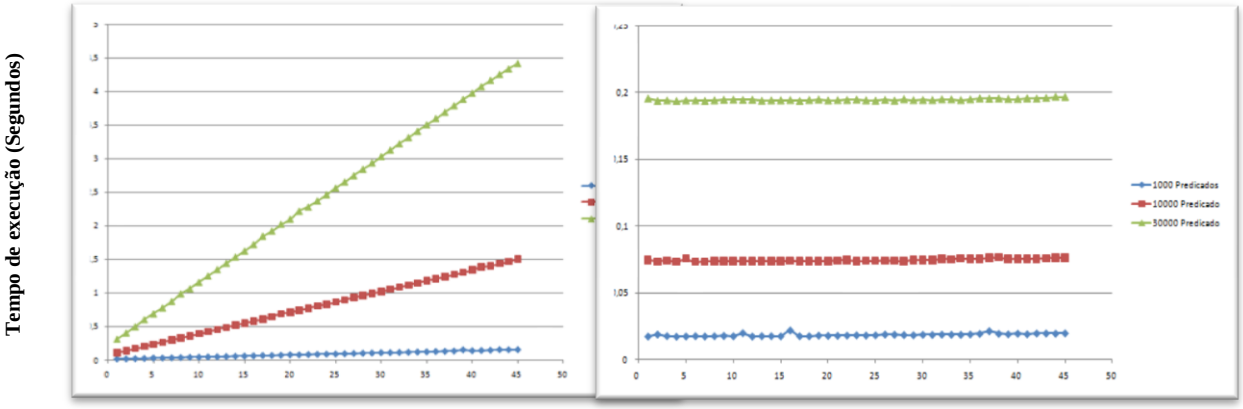
30000 Predicados		10000 Predicados		1000 Predicados	
Global (segundos)	Constante (segundos)	Global (segundos)	Constante (segundos)	Global (segundos)	Constante (segundos)
0,2171	0,0238	0,0772	0,0137	0,0146	0,0079
100%	9x	100%	5.6x	100%	1.8x

Tabela 4.7- Melhores resultados no *device 2*

Na tabela, a cinzento, estão os ganhos obtidos na utilização da *memória constante* para cada número diferente de predicados. Verifica-se um decréscimo no ganho explicado com a diminuição progressiva da dimensão do trabalho, e o progressivo impacto do lançamento dos *kernel* e *overhead* associado a uma granularidade de paralelização mais fina.

4.3.1.3. Variação do tamanho dos eventos no cálculo da *máscara*

Para o teste foi só utilizada uma configuração, que correspondeu a (6 *blocos*,128 *threads*). Nos gráficos seguinte estão representados todos os tempos obtidos para a *memória global* e *constante*:



a) **Número de atributos por Evento** **Número de atributos por Evento** b)

Figura 4.7 - Resultados obtidos na variação dos tamanhos dos eventos no cálculo da máscara
a) Global
b) Constante

Para a *memória global*, existe um aumento linear, ou seja, à medida que o número de atributos aumenta por evento, aumenta necessariamente o tempo para a obtenção da máscara.

Em relação à *memória constante* o tempo de obtenção da máscara é praticamente constante. Situando-se, em média, no valor 0,018 segundos. Isso pode ser explicado com o facto dos valores lidos da *memória global* serem pequenos, porque este teste foi apenas realizado para 1 evento, onde no máximo vai ter 1.2 *Kbytes* de tamanho (1 evento + 45 atributos = (45*4*3 + 8 bytes) + (45*4*4)).

Efectivamente, com estes resultados consegue-se observar que o ganho obtido com o uso da *memória constante* é muito grande. Por exemplo, com 45 atributos por evento, o tempo obtido na *memória global* foi **0,160923** segundos enquanto o tempo para *memória constante* foi **0,0199277**, **8x** de aumento.

No *device 2* os gráficos obtidos demonstram um comportamento ligeiramente diferente:

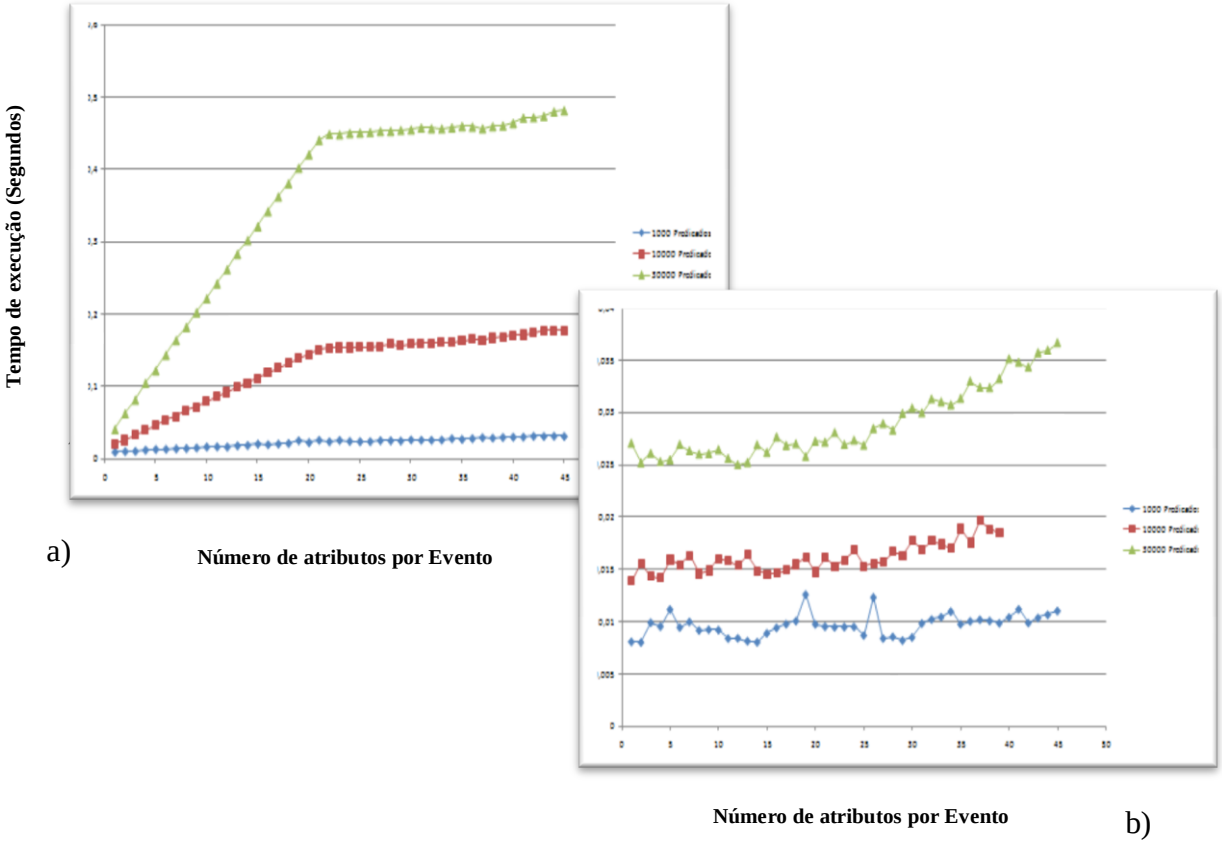


Figura 4.8 - Resultados obtidos na variação dos tamanhos dos eventos no cálculo da *máscara* para o *device 2*.

a) Global
b) Constante

Até sensivelmente 20 atributos por evento, o crescimento é linear mas, a partir desse ponto, existe uma estagnação e o declive da curva é menos acentuado. Tal pode ser explicado com o maior acesso à memória por parte dos 6 *SM* (dos 14 existentes) que estão a ser utilizados.

Como existem mais acessos por parte de todos os *threads* de cada *SM*, pode existir assim um aumento dos tempos por causa da largura de banda, ao contrário da execução com 1 *SM*, onde apenas no máximo 3 blocos podiam correr, não existindo esse problema.

Em relação à variação do número de predicados, o comportamento é análogo ao anteriormente apresentado, ou seja, à medida que aumenta o número de predicados o tempo cresce também, consequentemente.

4.3.2. Contagem

O número de registos para este *kernel* é menor, ou seja, 8, permitindo assim que o máximo número de *threads* permitido seja atingido, ou seja, 768.

De salientar que o estudo focou-se no impacto que a percentagem de predicados aceites tem no tempo para a obtenção das subscrições aceites. As percentagens variaram entre 10% e 100% de predicados aceites, em incrementos de 10%. O número de predicados foi sempre: 1000, 10000 e 30000 predicados no sistema. Os predicados aceites foram sempre definidos aleatoriamente através da função *rand()* do *ANSI-C*. As experiências também foram realizadas com as características indicadas na secção 4.1.

4.3.2.1. *Memória Global*

O gráfico obtido do débito (tempo de execução) da contagem para 30000 predicados foi:

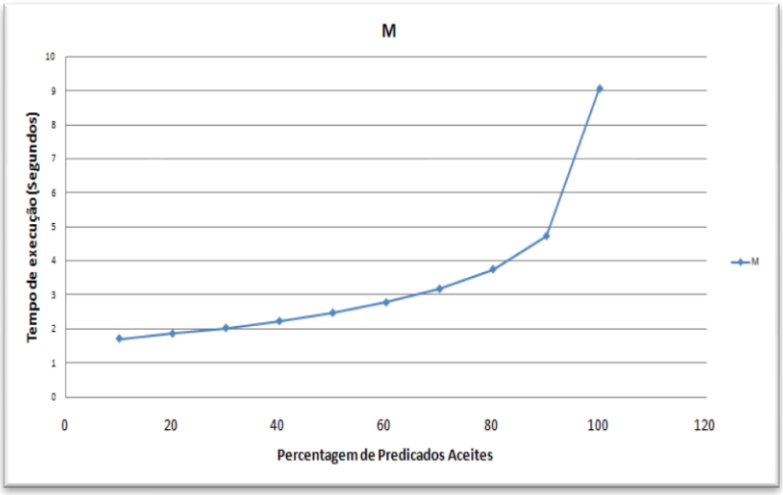


Figura 4.9 - Resultados do processo de contagem.

Através do gráfico verifica-se que, à medida que a percentagem de predicados aceites aumenta, o tempo de execução paralelamente também aumenta, provando que quantos mais predicados forem aceites mais tempo leva a execução. Em relação ao pico verificado na curva do ponto 90% para o 100%, pode ser explicado pelo facto de a probabilidade de serem aceites as subscrições com tamanho de 128 predicados (ou 64) ser igual a 100% no ponto 100 e portanto a

contagem percorre toda a estrutura, enquanto para o caso de 90%, a probabilidade de as subscrições falharem depende directamente da probabilidade de os predicados contidos nessas estarem aceites, e com probabilidades iguais ou inferiores a 90%, a probabilidade é menor. O comportamento verificado para 1000 predicados foi diferente.

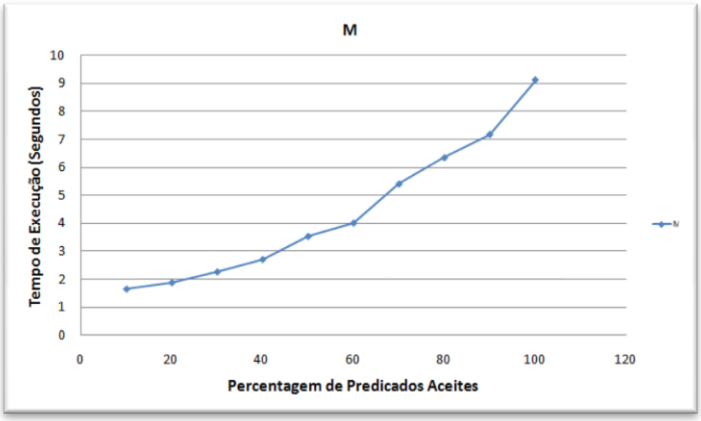


Figura 4.10 - Resultados do processo de contagem no *device 2*

Demonstra que a contagem no sistema tem maior probabilidade de percorrer todas as estruturas porque existem poucos predicados para um número grande de subscrições.

Seguidamente, é apresentada a tabela dos melhores resultados obtidos, para o pior caso, ou seja, para 30000 predicados (maior número de predicados) e 100% aceites. Os gráficos mostram que o número de predicados existentes no sistema quando são sempre satisfeitos não influencia o tempo para a contagem das subscrições, porque os valores obtidos estão próximos.

De relembrar que foram efectuadas 100 iterações para cada configuração definida (que estavam definidas entre 1…10 *blocos* e com 32…512 *threads* por *bloco*).

Percentagem	Tempos de Device 1 (M) (Segundos)	Tempos de Device 2 (GT) (Segundos)	Rácio (Aproximado)
10%	1,7102	0,2170	7.8x
20%	1,8565	0,2296	8x
30%	2,0255	0,2425	8.3x
40%	2,2256	0,2579	8.6x
50%	2,4686	0,2751	8.9x
60%	2,7741	0,2968	9x
70%	3,1741	0,3205	9.9x
80%	3,7546	0,3590	10x
90%	4,7326	0,4259	11x
100%	9,0846	0,6752	13x

Tabela 4.8 - Melhores resultados na contagem para ambos os *device*

A coluna *rácio* indica o aumento que existiu na utilização de um *device* com mais MP. Quanto maior fosse o trabalho, mais se notava o aumento do *rácio* dos resultados. A velocidade de *clock* dos multiprocessadores do *device 2* e o seu maior número explica nitidamente esses resultados. Verifica-se também que quanto maior for o trabalho, mais o *rácio* se aproxima do número de MP (14).

A seguir serão apresentados os resultados para a implementação que faz uso da *memória partilhada*, para uma análise comparativa e para avaliar o seu possível maior ganho.

4.3.2.2. *Memória Partilhada*

Os resultados obtidos em ambos os *devices* demonstram o mesmo comportamento que o verificado na *memória global*, mas com tempos efectivamente menores.

Mais uma vez, é apresentada uma tabela para o pior caso, ou seja 30000 predicados. Nela estão os aumentos obtidos com a utilização da *memória partilhada* em relação aos mesmos testes em *memória global.*

Percentagem	Tempos de Device 1 (M) (Segundos)	Aumento obtido com a utilização da memória partilhada
10%	1,5678	1.09x
20%	1,6498	1.12x
30%	1,7420	1.16x
40%	1,8570	1.19x
50%	1,9887	1.24x
60%	2,1647	1.28x
70%	2,3888	1.32x
80%	2,7164	1.38x
90%	3,2480	1.45x
100%	5,4546	1.66x

Tabela 4.9 - Melhores resultados na contagem com *memória partilhada* e ganhos obtidos no *device 1*

Existe pouco ganho em todos os casos, de aproximadamente 0.1 vezes mais do que na *memória global*. Na terceira coluna verificam-se os aumentos. Conclui-se também que, à medida que a percentagem aumenta (necessariamente também aumentam os acessos à *memória global*), há um aumento progressivo do *rácio*, porque esses acessos são evitados através da utilização da *memória partilhada*. O último caso sustenta tal constatação. Mas a sincronização associada a todos os *threads* no momento da leitura inicial pode estar a limitar os ganhos subjacentes.

A seguir será apresentado o possível ganho no *device 2*:

Percentagem	Tempos de Device 2 (GT) (Segundos)	Aumento obtido com a utilização da memória partilhada
10%	0,4038	0.53x
20%	0,4115	0.55x
30%	0,4183	0.57x
40%	0,4268	0.60x
50%	0,4377	0.62x
60%	0,4509	0.65x
70%	0,4677	0.68x
80%	0,4930	0.72x
90%	0,5321	0.80x
100%	0,6566	1.02x

Tabela 4.10 - Melhores resultados na contagem com *memória partilhada* e ganhos obtidos no *device 2*

Visivelmente, não houve um ganho, verificando-se até que existe uma perda para metade em alguns casos. Tal pode ser explicado pelo facto de todos os primeiros *threads* dos *blocos* escreverem *toda* a máscara para a *memória partilhada*, acrescentando assim um *overhead*. Esse é obtido porque todos os *threads* acedem ao mesmo tempo à *memória global*, verificando-se assim efeitos relacionados com a largura de banda, que não se verificavam na implementação da *memória global* porque a memória era acedida conforme a organização dos predicados nas subscrições. Tal constatação é reforçada com o último resultado, onde toda a máscara é acedida, e como tal apresenta um ganho, apesar de insignificante.

Como balanço, vai ser apresentada uma pequena tabela com os melhores resultados da contagem, para todos os números de predicados estudados, tanto *no device 1* como no *2*.

Através destes valores pode-se obter quantas subscrições podem ser obtidas, por segundo, neste sistema. De salientar que foram feitas 100 iterações por cada 500.000 subscrições, o que perfaz 50 milhões de subscrições. A unidade do débito é igual ao número de subscrições por segundo que são filtradas. No caso, os valores são sempre na ordem dos milhões. O tempo é em segundos.

	Device 1		Device 2	
Número de predicados	Tempo	Débito	Tempo	Débito
30000	5,4546	9,2M/S[3]	0,6566	76 M/S[3]
10000	5,1591	9,7 M/S[3]	0,5036	99 M/S[3]
1000	5,0281	9,9 M/S[3]	0,4379	114 M/S[3]

Tabela 4.11 - Melhores resultados na contagem para todos os números de predicados

[3] Milhões de subscrições por segundo

4.4. 32 Eventos

Os testes realizados foram os mesmos que os indicados na secção da implementação de 1 evento. Resumidamente, numa fase inicial, o estudo dos resultados incide mais no impacto que as configurações possíveis de execução de cada *kernel*, ou seja, o *tamanho da experiência*, têm na invocação de cada *kernel*, tanto na obtenção da *máscara de bits* como na contagem.

Para a primeira fase, o cálculo da *máscara*, as memórias *global* e *constante* foram testadas.

Em relação ao processo de contagem, o estudo foi efectuado apenas na *memória global*, porque a implementação deste processo não permitia a redução das estruturas para um tamanho suficientemente pequeno para ser guardado na *memória partilhada.* Já que cada inteiro da máscara representava 32 eventos e não podia haver aproveitamento dos *bits* como se verificou na solução anterior. Para todos os testes, o número de predicados também foi: 1000, 10000 e 30000.

De seguida, é apresentada a tabela com toda a informação relevante dos *kernel*:

Número de *threads* por *bloco*	Número de registos por *bloco*	Número máximo de *threads* activos	Melhor número de *blocos*
32	2048	128	4
64	2048	256	4
128	3840	256	2
256	7424	256	1

Tabela 4.12 - Melhores configurações do *kernel*

Nas subsecções seguintes vão ser apresentados os resultados do cálculo da *máscara*, tanto na *memória global* como na *constante*, e por último os testes realizados para o processo de contagem das subscrições para os 32 novos eventos.

4.4.1. Cálculo da *Máscara de Bits*

A obtenção dos eventos e predicados segue o mesmo mecanismo que o apresentado para 1 evento, apenas existindo a diferença que são gerados mais 31 eventos aleatoriamente.

4.4.1.1. *Memória global*

O *kernel*, que calcula a máscara, necessita de 29 registos por *threads* neste teste. Assim, perfaz as seguintes características para cada *tamanho* do *bloco:*

Kernel	Número de Registos por *thread*
Cálculo Máscara de bits (Global)	29
Cálculo Máscara de bits (Constante)	22
Contagem (Global)	9

Tabela 4.13 – Número de registos por *kernel*

Os resultados obtidos permitiram concluir que o mesmo comportamento da implementação de 1 evento é verificado, apenas variando os tempos de execução. O declive da recta dos resultados não é tão acentuado porque o *overhead* da comutação dos *threads* é "escondido" pelo tempo de processamento do trabalho, que se verificou muito maior.

A título de exemplo, o gráfico para 30000 predicados no *device 1* foi:

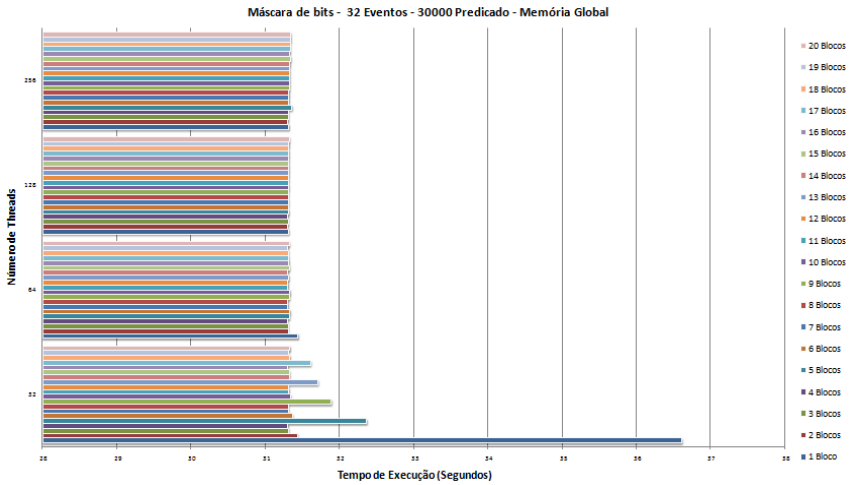


Figura 4.11 - Resultados do cálculo da *máscara* para 30000 predicados e 32 eventos.

Os resultados por entre as configurações não variam. Esse facto é reforçado por este *kernel* necessitar de 29 registos por *thread*, o que permite correr no máximo um número limitado de *threads* por *bloco*, variando consoante o tamanho do *bloco*. Esse número revelou-se menor porque os registos necessários também são em maior número.

A seguir os resultados obtidos são apresentados juntamente com as conclusões acerca das configurações:

30000 Predicados		10000 Predicados		1000 Predicados	
Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração
31,3015	(4, 64)	10,4523	(10, 64)	1,0594	(2, 32)

Tabela 4.14 – Melhores resultados obtidos para *device 1*

A sombreado estão os resultados que efectivamente provam que os melhores tempos são obtidos através do maior número de *threads activos* permitido pelos registos, ou então, são um número múltiplo desse valor máximo.

Para o *device 2* os melhores tempos foram:

30000 Predicados		10000 Predicados		1000 Predicados	
Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração
2,0985	(13, 256)	0,7364	(13, 256)	0,0923	(19, 32)

Tabela 4.15 – Melhores resultados obtidos para *device 2*

Neste exemplo, nos dois primeiros casos, os valores obtidos são ligeiramente melhores do que os obtidos na suposta melhor configuração, ou seja, com 14 blocos. No primeiro caso há apenas uma diferença de 5%, enquanto no exemplo para 10000 predicados a diferença foi de 12%. Para a configuração com o máximo de 32 *blocos* foi verificado que quantos mais *blocos* fossem melhores tempos eram obtidos.

4.4.1.2. *Memória Constante*

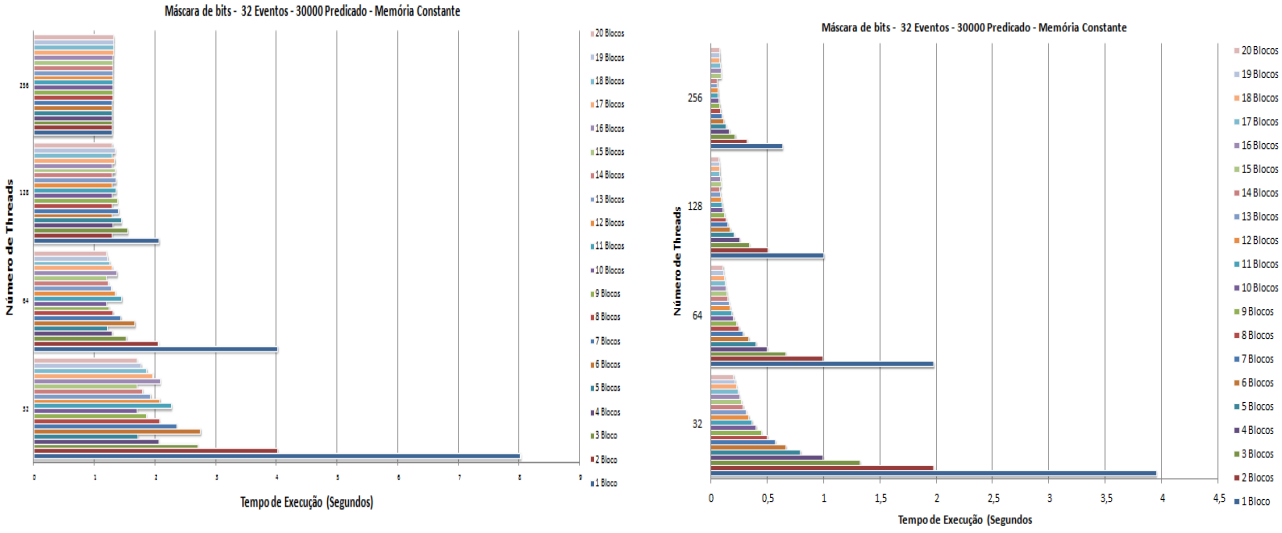
Para a implementação da *memória constante*, os registos necessários por cada *thread* variaram. Neste exemplo eram necessários 22 registos, logo, a tabela que indica a melhor configuração é:

Número de *threads* por *bloco*	Número de registos por *bloco*	Número máximo de *threads* activos	Melhor número de *blocos*
32	1536	160	5
64	1536	320	5
128	2816	256	2
256	5632	256	1

Tabela 4.16 – Melhores configurações para o *kernel*

Os resultados obtidos permitiram, mais uma vez, concluir que o mesmo comportamento da implementação de 1 evento é verificado, apenas variando os tempos de execução. Efectivamente nos gráficos obtidos verificam-se vários picos. Esses picos são explicados por serem sempre os *blocos* seguintes ao *bloco* múltiplo do maior número possíveis de estarem *activos.*

No gráfico seguinte, para 30000 predicados e *device 1* e *2*, esse comportamento é claramente verificado:



a) b)

Figura 4.12- Resultados do cálculo da *máscara* para 30000 predicados e 32 eventos com memória partilhada.

a) *device 1*
b) *device 2*

No *device 2,* os resultados verificados foram iguais em todos os casos. O comportamento verifica-se porque o melhor número de *threads activos* são sempre obtidos em todas as configurações. Como os *blocos* são distribuídos pelos multiprocessadores disponíveis, cada um corre todos os que recebe por conterem, sempre, o maior número de *threads activos*, ou pelo menos, números múltiplos de *threads* possíveis de correr, como se pode verificar na tabela. No mínimo pode correr 1 *bloco* por execução.

Em relação aos melhores resultados obtidos para cada *device*, foram os seguintes:

30000 Predicados		10000 Predicados		1000 Predicados	
Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração
1,1962	(10, 64)	0,4140	(5, 64)	0,0594	(1, 32)

Tabela 4.17 – Melhores resultados para todos os predicados no *device 1*

30000 Predicados		10000 Predicados		1000 Predicados	
Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração	Melhor Tempo (Segundos)	Melhor Configuração
0,0617	(14, 256)	0,0278	(14, 256)	0,0134	(16, 64)

Tabela 4.18 – Melhores resultados para todos os predicados no *device 2*

A sombreado estão, mais uma vez, os resultados que efectivamente mostram que os melhores tempos são obtidos através do maior número de *threads activos* permitido pelos registos.

Apenas no caso de 1000 predicados do *device 2*, não se comprova que a melhor configuração é obtida através da fórmula fornecida pelo fabricante proprietário. Como no teste da *memória global*, os melhores resultados são obtidos quantos mais *blocos* forem lançados.

De seguida será feita a análise de qual foi o ganho efectivo obtido através do uso da *memória constante* neste processo de preenchimento da *máscara de bits*.

Device 1								
30000 Predicados			10000 Predicados			1000 Predicados		
Global	Constante	Ganho	Global	Constante	Ganho	Global	Constante	Ganho
31,3015	1,1962	**26.2x**	10,4523	0,4141	**25.2x**	1,0594	0,0594	**17.8x**
Device 2								
30000 Predicados			10000 Predicados			1000 Predicados		
Global	Constante	Ganho	Global	Constante	Ganho	Global	Constante	Ganho
2,09849	0,0617	**33.9x**	0,736379	0,0278	**26.5x**	0,0923	0,0134	**6.9x**

Tabela 4.19 – Ganhos obtidos com a utilização da *memória constante*, em ambos os *device*.

Verifica-se que existe um ganho considerável utilizando a *memória constante*. Em todos os casos houve ganhos, em média o valor foi **22.75x** mais rápido. Pode-se concluir que a utilização da *memória constante*, para a manutenção das estruturas dos novos eventos que entram no sistema é a melhor escolha. Assim o tempo de cálculo da *máscara de bits*, em ambos os *devices* é melhor.

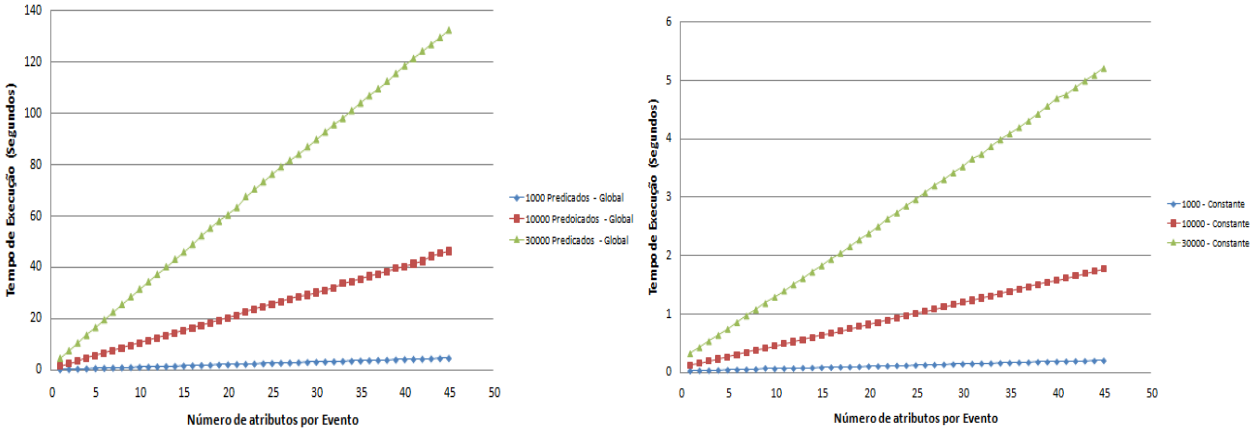
4.4.1.3. Variação do tamanho dos eventos no cálculo da *máscara*

Nesta secção, foi estudado o peso que o tamanho dos eventos tem no cálculo da *máscara*. Neste caso todos os 32 eventos terão entre 1 e 45 atributos, sendo que todos são percorridos por cada predicado.

Efectivamente, o comportamento verificado foi igual ao que se observou para 1 evento. Mais uma vez, à medida que o tamanho dos eventos é maior, maior se torna o tempo de execução. Em relação à implementação utilizando a *memória constante*, os tempos já não se revelaram tão constantes e a recta teve um ligeiro declive à medida que os eventos se tornavam

maiores. Pode-se explicar isso pelo facto do volume de dados, associado aos 32 eventos processados pelo *kernel*, ser bastante maior e, como tal, os dados mantidos em *memória constante* podem ser maiores do que os 8 *KBytes* possíveis de se manterem em *cache.*

No gráfico seguinte, que apresenta os três resultados para os predicados diferentes, constata tal facto:



a) b)

Figura 4.13 - Resultados obtidos no cálculo da máscara com a variação do tamanho dos eventos.

c) *global*
d) *constante*

A conclusão retirada destes resultados é que o débito associado à obtenção da *máscara* depende *directamente* do número de predicados existentes no sistema e do número de atributos que compõem os eventos, ou seja, quanto mais predicados e atributos (nos eventos) existirem, mais tempo é necessário para o processo de obtenção da *máscara de bits*. É uma conclusão que facilmente era deduzida já que, quanto maior forem os eventos, mais trabalho tem que ser processado.

4.4.2. Contagem

Aqui, apenas foi utilizada a *memória global* porque toda a informação dos predicados está indexada em vários inteiros, no caso, eram ou 1000, ou 10000 ou 30000 predicados. Em todos os casos apenas o primeiro era possível de ser representado em *memória partilhada*, mas como o trabalho era tão pequeno, os possíveis ganhos não compensavam a complexidade do processo. Relembrando, a *memória partilhada* por multiprocessador é de apenas 16 *KBytes*.

Na secção seguinte serão então apresentados os resultados obtidos em ambos os *devices* e mais uma vez para o número de predicados indicado em cima.

4.4.2.1. *Memória Global*

Neste teste a percentagem de predicados aceites variou de 10% até 100%. De recordar que neste novo *kernel*, para 32 eventos, os acessos são feitos como se de um evento se tratasse, ou seja, um inteiro (que representa um predicado) na *máscara de bits* indica o resultado da aceitação para 32 eventos de uma vez, bastando efectuar um *and* lógico dos *bits*.

O comportamento verificado dos resultados foi, mais uma vez, semelhante ao obtido para 1 evento, muito devido à própria implementação do *kernel*.

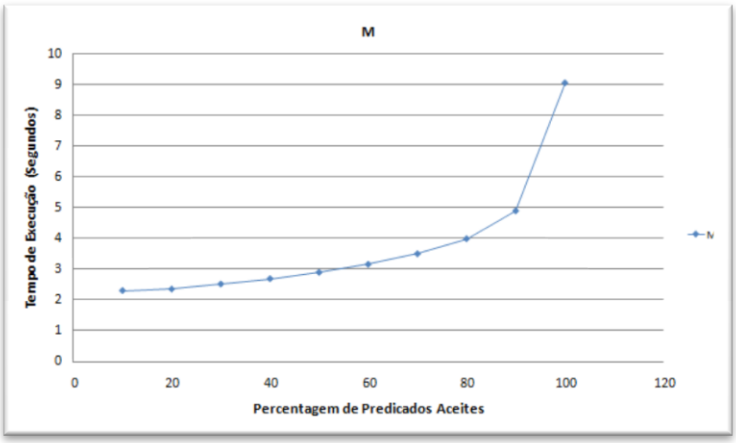


Figura 4.14 - Resultados obtidos no processo de contagem.

Relembrando as causas, até aos 90% de predicados aceites, as subscrições de 128 e de até 64 predicados têm grande probabilidade de falharem antes da verificação total dos predicados, acrescentando assim um ganho adjacente que não se verifica para os 100%, daí a grande diferença de resultados.

A tabela, para o pior caso, ou seja 30000 predicados, vai ser apresentada de seguida. Nela consta apenas o *rácio* obtido através do uso do *device* com mais multiprocessadores.

Perc.	Tempos de Device 1 (M) (Segundos)	Tempos de Device 2 (GT) (Segundos)	Rácio (Arredondado)
10%	2,3001	0,2514	9.1x
20%	2,3482	0,2619	8.9x
30%	2,5136	0,2713	9.3x
40%	2,6858	0,2805	9.6x
50%	2,8921	0,2989	9.7x
60%	3,1550	0,3237	9.7x
70%	3,4963	0,3481	10x
80%	3,9872	0,3707	10.7x
90%	4,8990	0,4324	11.3x
100%	9,0690	0,6673	13.6x

Tabela 4.20 – Melhores tempos obtidos no processo de contagem de ambos os *devices* e o ganho obtido com a utilização do *device 2*.

Os *rácios* obtidos revelaram-se bastante grandes. Verifica-se que efectivamente uma placa com mais multiprocessadores e memória dedicada consegue obter melhores resultados. A diferença do número de multiprocessadores também é mais notada à medida que o trabalho aumenta, comprovando-se com a tendência do *rácio* estar a convergir para 14 vezes mais.

Como balanço, vai ser apresentada uma pequena tabela com os melhores resultados da contagem para ambos os *device:*

	Device 1		Device 2	
Número de predicados	Tempo	Débito	Tempo	Débito
30000	9,0690	5,5 M/S	0,6673	75 M/S
10000	8,9834	5,6 M/S	0,6546	76 M/S
1000	9,0756	5,5 M/S	0,6614	76 M/S

Tabela 4.21 – Melhores tempos e débito para ambos os *device.*

Os valores apresentados são aproximados, mas, revelam que o débito do sistema é bastante alto, processando por segundo aproximadamente 5 milhões de subscrições. No *device 2*, esse débito já é de aproximadamente 76 milhões de subscrições por segundo. Efectivamente é um valor bastante elevado, visto ser obtido para 32 eventos de uma vez.

4.5. Resultados no *CPU (Host)*

Foram realizados os mesmos testes, tanto o cálculo da *máscara*, como o processo de contagem. Também foram realizadas 100 iterações em cada processo e a geração dos eventos e predicados para a máscara foi utilizando o mesmo método. Portanto, os eventos e predicados foram iguais aos testes do *GPU*, tal como as subscrições.

Os testes para a obtenção da máscara foram efectuados com eventos de tamanhos diferentes. Essa variação mais uma vez foi entre 1 a 45 atributos. Em relação ao processo de contagem, foi realizado para várias percentagens de predicados aceites como foi efectuado no *device*. A variação em ambos os testes dos predicados foi de 1000, 10000 e 30000 predicados.

Os resultados apresentados do cálculo da máscara são apenas o teste de 10 atributos por evento, como aconteceu com os testes no *GPU*. Os resultados apresentados da contagem são apenas os do pior caso (100% e 30000 pred.), de forma a poder-se comparar com os anteriormente apresentados.

30000 Predicados				10000 Predicados				1000 Predicados			
1 Evento		32 Eventos		1 Evento		32 Eventos		1 Evento		32 Eventos	
Máscara	Contagem	Máscara	Contagem	Máscara	Contagem	Máscara	Contagem	Máscara	Contagem	Máscara	Contagem
0,1620	1,8509	6,2400	2,0968	0,0810	1,4960	2,0970	1,8560	0,005	1,3391	0,2050	1,5545

Tabela 4.22 – Melhores tempos obtidos no CPU para o cálculo da máscara e contagem.

4.6. Conclusões retiradas do processo de Filtragem

Em todas as implementações os resultados revelaram-se bastante promissores. Mas, ambos os testes não podem ser somados e obter-se um resultado que indique o tempo de filtragem total das subscrições. Para tal efeito, realizou-se mais um pequeno teste onde os predicados, no cálculo da *máscara* e na contagem, são todos aceites. Tal foi possível definindo todos os elementos predicados da seguinte forma: atributo igual a 0, operador de comparação igual a = e valores iguais a 1. Os eventos teriam o mesmo atributo (0) e o mesmo valor (1). Foi utilizada a implementação para *memória constante* em ambos os *device.* A configuração foi 6 *blocos* e 128 *threads.*

Seguidamente, é apresentada a tabela geral com os resultados obtidos nas três unidades de processamento do estudo. Relembrar que foram realizadas 100 iterações para 500 mil subscrições, o que equivale a ter um processo para 50 milhões de subscrições.

Na tabela o débito representa o fluxo de subscrições aceites por segundo, para 32 eventos. No teste de 1 evento o tempo total da filtragem foi multiplicado por 32 para que os valores estejam nas mesmas unidades. De realçar que essa decisão não está a contabilizar o *overhead* acrescido das várias chamadas dos métodos da implementação para 1 evento que ainda acresciam o tempo de execução.

	Device 1							
	1 Evento				32 Eventos			
Número de predicados	Máscara	Contagem	Total filtragem	Débito (32e)	Máscara	Contagem	Total filtragem	Débito (32e)
30000	0,2127	5,4546	5,6670	**276 k/S**[4]	8,7386	9,0690	17,8076	**3 M/S**
10000	0,0818	5,1592	5,2409	**298 k/S**[4]	2,9645	8,9834	11,9479	**4 M/S**
1000	0,0198	5,0281	5,0479	**310 k/S**[4]	0,3123	9,0756	9,3880	**5 M/S**
	Device 2							
	1 Evento				32 Eventos			
30000	0,0330	0,6566	0,6897	**2 M/S**	1,3342	0,667364	2,0015	**25 M/S**
10000	0,0165	0,5036	0,5201	**3 M/S**	0,4725	0,654679	1,1271	**44 M/S**
1000	0,0095	0,4379	0,4474	**3 M/S**	0,0753	0,661499	0,7367	**68 M/S**
	CPU							
	1 Evento				32 Eventos			
30000	0,7886	1,8509	2,639	**592 k/S**[4]	26,0551	2,0968	28,1519	**2 M/S**
10000	0,2516	1,4960	1,7477	**894 k/S**[4]	8,7857	1,8560	10,6418	**5 M/S**
1000	0,0253	1,3391	1,3644	**1 M/S**	0,8581	1,5545	2,4127	**21 M/S**

Tabela 4.23 – Todos os débitos das unidades de processamento para o processo de filtragem com todos os números de predicados mas com todos aceites.

[4] Milhares de subscrições por segundo

Os resultados sustentam claramente que o *GPU* é uma boa escolha para este processo. O débito obtido no *device 2* é bastante grande e demonstra as grandes potencialidades destas unidades de processamento. Com estes resultados está também provado o ganho obtido com a implementação para 32 eventos, porque como se verifica, o débito obtido é bem maior em todos os casos do que para 1 evento.

Os resultados obtidos no *CPU* indiciam novamente que a opção *GPU* é bastante viável. Verificando o débito obtido no *CPU* testado com o *device* com menos poder, os ganhos não são significativos (apenas melhor em um resultado), mas, comparando com o segundo *device* estes conseguem-se aproximar a um valor **10 vezes mais rápidos** em alguns casos, por exemplo, o de 32 eventos é demonstrativo.

5. Conclusão

5.1. Apreciações Finais

Em sistemas editor/assinante baseados no conteúdo, o processo da filtragem é um ponto fulcral, em especial nos sistemas baseados em *brokers* onde o número de assinantes é usualmente muito significativo. Como o número de subscrições médio é proporcional ao número de assinantes, e como a taxa de entrada de novos eventos nestes sistemas é geralmente muita elevada, para os *brokers* torna-se importante a eficiência e rapidez de obtenção do resultado da filtragem. Essa eficiência pode ser adquirida através da utilização de um algoritmo capaz de obter tais requisitos. Nesta dissertação um algoritmo foi escolhido [17] para esse processo, por ter as melhores características para tal fim e também porque era o que tinha as melhores características para ser paralelizado nos *GPUs*.

Os *GPUs* foram as unidades de processamento escolhidas para esse trabalho, já que hoje em dia revelam ser uma boa escolha na relação qualidade/preço, porque com a convergência da sua arquitectura para um modelo mais paralelo, as possibilidades de se retirar grande proveito destas unidades aumentam. Juntamente com o aparecimento de novas paradigmas de programação [7], tornaram-se realmente numa solução aliciante a explorar.

Foi verificado nesta dissertação que são efectivamente boas escolhas para este domínio aplicacional, porque os resultados obtidos demonstraram que são equiparáveis ou melhores que os *CPUs comuns*. Revelaram também que mais soluções são possíveis, como dividir o trabalho por mais *GPUs* ou uma solução híbrida entre *CPU+GPU*.

As principais contribuições obtidas foram as seguintes:

- Foi produzido um estudo e uma análise da viabilidade dos *GPUs* para o processamento de dados em sistemas editor/assinante *baseados no conteúdo*, que se revelou bastante positivo.
- Foram estudadas as adaptações efectuadas ao algoritmo escolhido como referência, que se revelaram bastante promissoras.
- Foi efectuado um estudo experimental e crítico da implementação, que verificou a viabilidade dos *GPUs* neste contexto.

5.2. Trabalho Futuro

O sistema desenvolvido oferece bastantes possibilidades de extensões para serem tratadas no futuro. Nesta secção apresentam-se as que parecem mais importantes.

A exploração de *GPUs* com mais *SM* ou a utilização conjunta de vários *GPUs*, como por exemplo em *SLi*, seria uma solução a explorar porque poderia tirar partido de mais memórias e de mais multiprocessadores disponíveis.

Algumas características de mais baixo-nível que não foram exploradas, relacionadas com o *GPUs* foram:

- Divisão do trabalho por *half-warps* e utilizando *blocos* de *threads* a 2 e 3 dimensões.
- Aumentar a coerência dos acessos à memória e tirar melhor partido dos vários tipos de memória existentes nos *GPUs*, em particular as memórias optimizadas para leitura.
- Utilizar primitivas de comunicação assíncrona entre o *CPU* e o *GPU*.

A possibilidade suplementar de execução conjunta entre o *CPU* e o *GPU* também podia ser explorada. O trabalho mais repetitivo e linear podia ser processado no *GPU*, como por exemplo a contagem, e o *CPU* obtinha a *máscara*, de modo a rentabilizar o uso conjunto de ambas as *unidades*. Existe assim, uma maior probabilidade de aumento do débito de resultados do sistema, porque a obtenção e processamento do trabalho são feitos em paralelo, por ambas as *unidades*. Para isso, uma extensão do CUDA, *mCUDA* [23], podia ser utilizada, pois é capaz de tirar partido da paralelização tanto nos *GPUs* como nos *CPUs* de uma forma mais directa. Em alternativa e de modo a conseguir uma solução mais abrangente, existe a possibilidade de transpor a implementação para a plataforma *OpenCL* [32,33]. Teria a vantagem de combinar melhor o trabalho do *CPU* e do *GPU* porque tem um modelo de programação mais uniforme. A possibilidade do trabalho correr noutras placas gráficas, que não *NVIDIA*, também seria um ganho significativo.

O tratamento dos ficheiros *RSS* e das expressões *XPATH* é outra questão que necessita de mais atenção. A implementação desenvolvida trata o processo de filtragem de forma ainda abstracta, usando em termos dos elementos base subscrições, predicados e eventos.

Finalmente, outra linha de trabalho poderia passar pela implementação do algoritmo [18] que foi mencionado como um forte candidato, mas que, por motivos de falta de tempo não foi considerado.

6. Bibliografia

6.1. Bibliografia Principal

[1] L. Hongzhou V. Ramasubramanian, E. G. Sirer, department of Computer Science, Cornell University, Ithaca, NY 14853, Client Behavior and Feed Characteristics of RSS, a Publish-Subscribe System for Web Micronews, September 2005.

[2] Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for contentbased networking. In INFOCOM, 2004.

[3] J. L. Martins and S. Duarte. Routing Algorithms for Content-based Publish/Subscribe Systems. IEEE Communications Tutorials and Surveys – Accepted for Publication, page 21, 2009.

[4] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.1, 2008.

[5] Rose, R. Murty, P.Pietzuch, et al. Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds. In Proc. Of NSDI, Cambridge, MA, USA, 2007.

[6] N. Rubin, 2008. GPU evolution: will graphics morph into compute?. In Proceedings of the 17th international Conference on Parallel Architectures and Compilation Techniques (Toronto, Ontario, Canada, October 25 - 29, 2008). PACT '08. ACM, New York, NY, 1-1.

[7] M. Harris, D. Luebke, I. Buck, N. N. Govindaraju, J. Kruger, A. E. Lefohn, T.J. Purcell, C. Wooley, 2004. GPGPU: General-purpose computation on graphics hardware. Course notes 32 of SIGGRAPH 2004.

[8] D. Kirk/NVIDIA and W. W. Hwu, Programming massively parallel processors, ECE 498AL1, University of Illinois, Urbana-Champaign, Fall 2007

[9] C.Y. Chan , P. Felber , M. Garofalakis , R. Rastogi, Efficient filtering of XML documents with XPath expressions, The VLDB Journal — The International Journal on Very Large Data Bases, v.11 n.4, p.354-379, December 2002

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W. W. and Hwu, 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Salt Lake City, UT, USA, February 20 - 23, 2008). PPoPP '08. ACM, New York, NY.

[11] Y. Yuan and M.J. Shaw, Induction of fuzzy decision trees. Fuzzy Sets and Systems 69 (1995), pp. 125–139

[12] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Transaction on Computers, 35(8):677–691, 1986.

[13] M. K. Aguilera , R. E. Strom , D. C. Sturman , M. Astley , T. D. Chandra, Matching events in a content-based subscription system, Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing, p.53-61, May 04-06, 1999, Atlanta, Georgia, United States

[14] S. Tarkoma, Chained forests for fast subsumption matching, Proceedings of the 2007 inaugural international conference on Distributed event-based systems, June 20-22, 2007, Toronto, Ontario, Canada

[15] Carzaniga, D. S. Rosenblum, and A. L. Wolf, 2001. Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst. 19, 3 (Aug. 2001), 332-383.

[16] Campailla , S. Chaki , E. Clarke , S. Jha , H. Veith, Efficient filtering in publish-subscribe systems using binary decision diagrams, Proceedings of the 23rd International Conference on Software Engineering, p.443-452, May 12-19, 2001, Toronto, Ontario, Canada

[17] F. Fabret , H. A. Jacobsen , F. Llirbat , J. Pereira , K. A. Ross , D. Shasha, Filtering algorithms and implementation for very fast publish/subscribe systems, Proceedings of the 2001 ACM SIGMOD international conference on Management of data, p.115-126, May 21-24, 2001, Santa Barbara, California, United States

[18] G. Ashayer , H. Ka Yau Leung , H. Arno Jacobsen, Predicate Matching and Subscription Matching in Publish/Subscribe Systems, Proceedings of the 22nd International Conference on Distributed Computing Systems, p.539-548, July 02-05, 2002

[19] M. Altinel and M.J. Franklin, 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th international Conference on Very Large Data Bases* (September 10 - 14, 2000). A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K. Whang, Eds. Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, 53-64.

[20] Y. Diao, M. Franklin, "High-Performance XML Filtering: An Overview of YFilter", ICDE, 2003.

[21] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P. Fischer. 2003. Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. Database Syst. 28, 4 (Dec. 2003), 467-516.

[22] Y. Diao, S. Rizvi, and M. J. Franklin, 2004. Towards an internet-scale XML dissemination service. In Proceedings of the Thirtieth international Conference on Very Large Data Bases - Volume 30 (Toronto, Canada, August 31 - September 03, 2004). M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, Eds. Very Large Data Bases. VLDB Endowment, 612-623.

[23] J. Stratton, S. Stone, W. Hwu. "MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores". Technical report, University of Illinois at Urbana-Champaign, IMPACT-08-01, March, 2008.

6.2. Bibliografia Suplementar

[24] UserLand. RSS 2.0 Specifications. http://blogs.law.harvard.edu/tech/rss, 2003

[25] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, eXtensible Mark-up Language (XML) 1.0 Specifications. http://www.w3.org/TR/2006/REC-xml20060816/, 2006

[26] World Wide Web Consortium. W3C. http://www.w3c.org, 2009

[27] Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie, J. Siméon, XML path language (XPath) 2.0 W3C working draft 16, Tech. Rep. WDxpath20-20020816, World Wide Web Consortium, August 2002. http://www.w3.org/TR/xpath20

[28] What is GPU?. http://www.nvidia.com/content/nsist/module/what_gpu.asp, 2006

[29] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, XQuery 1.0: An XML Query Language, World Wide Web Consortium, January 2007, http://www.w3.org/TR/xquery/

[30] Shaders, http://en.wikipedia.org/wiki/Shaders, 2009.

[31] CUDA, Compute Unified Device Architecture, http://www.nvidia.com/object/cuda_home.html, 2009.

[32] OpenCL, http://www.khronos.org/opencl/, December 2008

[33] Munshi, OpenCL: Parallel Computing on the GPU and CPU, SigGraph2008

[34] Direct3D 11, http://www.xnagamefest.com/presentations08.htm, 200

[35] D. Megginson. SAX: A Simple API for XML, http://www.megginson.com/SAX/, 2009

[36] Folding@Home website. http://folding.stanford.edu, 2007.